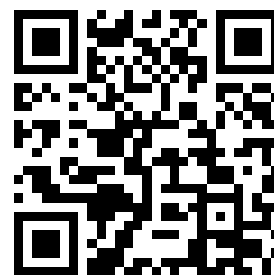

This is a reproduction of a library book that was digitized by Google as part of an ongoing effort to preserve the information in books and make it universally accessible.

GoogleTM books

<https://books.google.com>



510.78 U62 U.S. Symposium on advanced programming...

510.78 .U62 C.1
Symposium on advanced
Stanford University Libraries



3 6105 046 430 711

SAL

SAL3

STANFORD UNIVERSITY
COMPUTER SCIENCE
LIBRARY



STANFORD
LIBRARIES

1953

symposium on
ADVANCED PROGRAMMING
METHODS FOR
DIGITAL COMPUTERS

WASHINGTON, D.C.
JUNE 28,29,1956

under the joint sponsorship of
Navy Mathematical Computing Advisory Panel
and
Office of Naval Research

Office of Naval Research • Department of the Navy

proceedings

EX LIBRIS

STANFORD



LIBRARIES

Gift of

Mr. and Mrs. Julian J. Meyer

STANFORD UNIVERSITY
JAN 1958

ONR Symposium Report
ACR-15

symposium on LIBRARY
**ADVANCED PROGRAMMING
METHODS FOR
DIGITAL COMPUTERS**

STANFORD UNIVERSITY
JAN 1958
LIBRARY

**WASHINGTON, D.C.
JUNE 28,29,1956**

under the joint sponsorship of
U.S. **Navy Mathematical Computing Advisory Panel**
and
Office of Naval Research

Office of Naval Research • Department of the Navy

Digitized by Google

510.72

U62

1921

PREFACE

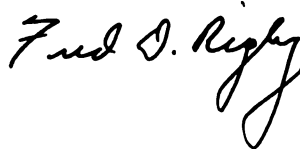
The rapid expansion of computing activities since the last symposium sponsored by the Navy Mathematical Computing Advisory Panel in 1954 has emphasized a need to assess and evaluate the impact that advanced programming methods have had on the operation of computing installations during this period. Consequently, this Panel organized a symposium which was held 28 and 29 June, 1956 and was attended by over two hundred and fifty persons representing agencies of the Federal Government and its contractors.

The choice of speakers at the symposium was made with the intent of furnishing a picture, necessarily incomplete, of present as well as future developments in this active area. Any complete representation of those persons in government, universities, and industry who have made and are making significant contributions in this field would have been impossible and was not attempted.

The Office of Naval Research is especially grateful to the speakers for their presentations and to the following for their assistance in planning the symposium:

John W. Backus, International Business Machines Corporation
John W. Carr III, University of Michigan
Grace M. Hopper, Sperry Rand Corporation
Arthur S. Kranzley, Radio Corporation of America
Wesley S. Melahn, Lincoln Laboratory
John Pasta, Atomic Energy Commission
Morgan R. Walker, E. I. duPont de Nemours
Joseph H. Wegstein, National Bureau of Standards

Particular thanks are due Milton E. Rose of the Office of Naval Research who, in addition to his duties as Chairman of the Program Committee in coordinating the symposium, collected and edited the papers of this symposium. It is hoped they will serve as valuable reference material.



F. D. Rigby, Chairman
Mathematical Computing Advisory Panel

Office of Naval Research
Washington, D. C.
October 1956

CONTENTS

SESSION I - 13 JUNE 1956

Chairman: Fred D. Rigby, Office of Naval Research

THE INTERLUDE 1954 - 1956	1
Grace M. Hopper Remington Rand Univac Division Sperry Rand Corporation	
AUTOMATIC CODING PRINCIPLES	3
Joseph H. Wegstein National Bureau of Standards	
DEVELOPMENT OF COMMON LANGUAGE AUTOMATIC PROGRAMMING SYSTEMS	7
Charles E. Thompson Hanford Atomic Products Operation General Electric Company	
PRODUCTION OF LARGE COMPUTER PROGRAMS	15
H. D. Bennington Lincoln Laboratory Massachusetts Institute of Technology	
SHARE - A STUDY IN THE REDUCTION OF REDUNDANT PROGRAMMING EFFORT THROUGH THE PROMOTION OF INTER-INSTALLATION COMMUNICATION.	29
Fletcher Jones North American Aviation, Inc.	
ADVANCED PROGRAMMING TECHNIQUES WITH SMALLER COMPUTERS	35
John W. Carr, III, and B. Arden University of Michigan	

SESSION II - 14 JUNE 1956

Chairman: Albert E. Smith, Bureau of Ships

COMPUTING AT LOS ALAMOS, GROUP T-1	39
Max Goldstein University of California Los Alamos Scientific Laboratory	
CODING FOR THE MANIAC	45
Mark Wells University of California Los Alamos Scientific Laboratory	

PROPOSED ADVANCED CODING SYSTEM FOR UNIVAC-LARC	49
Frances E. Holberton	
David Taylor Model Basin	
RCA APPROACH TO AUTOMATIC PROGRAMMING FOR COMMERCIAL PROBLEMS	57
John H. Waite, Jr.	
Radio Corporation of America	
THE PACT COMPILER FOR THE 701	67
R. G. Selfridge	
U.S. Naval Ordnance Test Station	
AUTOMATIC DIGITAL ENCODING SYSTEM II	71
E. K. Blum	
U.S. Naval Ordnance Laboratory	
ON A PROPERTY OF NATURAL LANGUAGE AND ITS USE FOR THE DESIGN OF IMPROVED MACHINE LANGUAGE (ASSOCIATIVE MACHINE LANGUAGES).	77
Robert Serrell	
Radio Corporation of America	
David Sarnoff Research Center	

SESSION I - 13 JUNE 1956

Chairman: Fred D. Rigby

***Office of Naval Research
Washington, D. C.***

THE INTERLUDE 1954 - 1956

Grace M. Hopper

*Remington Rand Univac Division
Sperry Rand Corporation
Philadelphia, Pennsylvania*

I have been asked to bridge the gap between the previous symposium on Automatic Programming for Digital Computers sponsored by the Office of Naval Research on 13 and 14 May 1954 and our meeting here today. In this period of time a whole new field of effort has grown up, much of it based on the work reported then. It was the first meeting devoted to interpreters, compilers, and generators, a whole new family of programs; and most of the routines, being less than a year old, were still untried. Yet much of the groundwork for present developments is recorded in the proceedings of 1954.

Most of the applications then discussed were concerned with mathematical and scientific problems, but today's applications to data processing follow from these origins. The distinction between two classes of programmers, the professionals and the laymen, was clear at that time and the difference between the requirements for open-shop versus closed-shop operation was discussed.

A few definitions are necessary to discuss this carryover. The basic types of routines involved in automatic coding are: converters, interpreters, assemblers, compilers, and generators. According to the Association for Computing Machinery Glossary, these may be defined as follows:

A converter changes information from one notation to another. Thus it may convert alphanumeric data into numeric, decimal information into binary, or fixed-point into floating-point data.

A pseudocode is an arbitrary instruction code, independent of the hardware of the computer, which must be translated into computer code if it is to direct the computer through a series of operations.

An interpreter is an executive routine which, as the computation progresses, translates a program expressed in pseudocode into computer code and performs the indicated operations by means of subroutines as they are translated.

An assembler is an executive routine which translates a set of relatively coded parts of a problem into a complete running program.

A compiler is an executive routine which, before the desired computation is started, translates a program expressed in pseudocode into computer code (or another pseudocode). In accomplishing the translation, the compiler may be required to instruct the computer to decode the pseudocode, convert information, select subroutines and/or generators from a library, oversee the operation of generators, allocate storage, assemble subroutines and sections of relative coding, and produce a record of the operations it has performed.

A generator is an executive routine which produces subroutines or complete routines from parameters delivered to it and from stored skeletal coding.

The first examples of each of these types of automatic coding routines were discussed at the 1954 meeting. It was our privilege to listen to Stanley Gill who, in collaboration with Wilkes and Wheeler, wrote "The Preparation of Programs for an Electronic Digital Computer," the first book in this new field.

In the field of conversion and interpretation, Charles W. Adams described the M. I. T. routines of which the Comprehensive System represents the most complete conversion and

interpretive system ever developed, since it assists not only the original programming of a problem but also the debugging of the problem. The Summer Session computer, designed for laymen, was a "pseudocomputer" presented by means of an interpretive routine.

David E. Muller told us of the development of subroutines and of interpretively applied floating-decimal, double-precision, and complex-number routines for the Illiac at the University of Illinois; John W. Carr III of Michigan outlined the MAGIC system of coding for the MIDAC; and John Backus presented the Speedcoding interpretive system for the IBM 701. Thus a very complete report was presented on the then-existing interpretive techniques, the pioneering efforts in automatic coding.

The concept of generation of coding was discussed by Betty Holberton of David Taylor Model Basin whose Sort Generator had first run in 1951; John Waite of the Radio Corporation of America and Merritt Elmore of the University of California Radiation Laboratory presented Editing Generators. These reports on generators contained the basic material which would provide the foundation for automatic coding for data-processing problems.

Three compilers, still very new in 1954, were discussed at the meeting. The A-2 Compiler and the NYU Compiler, both for the Univac, were reported by Nora Moser of the Army Map Service and Roy Goldfinger, then at New York University. The first report on a compiler designed for a small computer was made by Hugh Livingston of the Burroughs Corporation. These reports spurred the development of compilers until today almost all computers are "equipped" with one or more compiling systems of varying degrees of ability and suited to various types of problems and users.

Papers were presented on four topics which were wholly new and these opened new paths for the applications of computers. The presentation by Charles Adams of a description of Laning and Zierler's system of algebraic pseudocoding for the Whirlwind computer led to the development of Boeing's BACAIC for the 701, FORTRAN for the 704, AT-3 for the Univac, and the Purdue System for the Datatron and indicated the need for far more effort in the area of algebraic translators.

Allen Keller, of the General Electric Company, described a compiler-simulator for the solution of problems on an IBM 701 at a distance. Here the subroutines correspond to physical components of a system rather than to mathematical functions, and the compilation unites them into a descriptive program for a particular physical system. Further work along these lines is needed.

Harry Kahrmanian described an analytical differentiator — the first routine instructing a computer to carry out analytical rather than arithmetic operations. Finally, and perhaps most important for its future applications, Saul Gorn outlined a universal code — a pseudocode so designed that, once it had been used to state a problem, the problem definition could be presented to compilers operating on different computers to produce machine coding for those computers.

The dangers of "too many subroutines" and "too complex pseudocodes" were pointed out as warnings to be heeded in future developments. "Too many subroutines" has been answered by the use of generators, especially in the field of preparing programs for data processing. The answer to the criticism of "two complex pseudocodes" has been the use of normal mathematical symbology and English words.

Clearly, most of the principles upon which the developments of the past two years have been based were presented and published as a result of the 1954 ONR conference. I am grateful for the opportunity to thank the Office of Naval Research for those tremendously useful proceedings which are the only printed source for all of this fundamental work. I am especially thankful for the inclusion of the discussions of the papers.

It is to be hoped that this meeting will encourage future developments in programming to the same extent that the last one did. That automatic coding is accepted and operating today is due in large part to the information and encouragement received at the previous symposium. I am sure we will receive the same encouragement and enlightenment today.

AUTOMATIC CODING PRINCIPLES

Joseph H. Wegstein

*National Bureau of Standards
Washington, D. C.*

Once upon a time, long ago, there were computing systems. On the one hand there were problems: some quite scientific that perhaps required the roots of equations, and some from business or control or scheduling such as how many cherry pies to bake tomorrow. On the other hand there was a computer, a black box from the mathematician's point of view, which contained charges or no charges where charges shouldn't be, and these were represented as zeros or ones, called bits or binary digits for convenience of description.

Then programmers came and it was their job to bridge the gap between the problem and the computer (Fig. 1). Because the base 2 is hard to use they marked the bits off by three's and said it was an octal system or by four's and called it hexadecimal or sexidecimal. Taking a dozen or so of these characters at a time, they called them words and the computer obliged by taking in or feeding out these words. Some computers even spoke directly in decimal words. This was not so long ago. Less than three years ago one prominent computer laboratory circulated instructions sternly warning programmers not to set control-counters and order-registers by tapping bits into the computer with a clip lead. Of course today this warning is as out of date as a warning in a hotel for guests with mustaches to refrain from lighting their cigars from the gas jets.

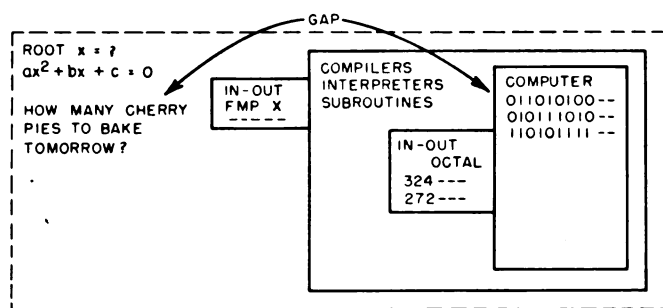


Fig. 1 - Computer systems schematic

Starting from the problem side of the gap, the programmer often had to do a further analysis of the problem. If the roots of $ax^2 + bx + c = 0$ were called for, he had to write that $x = (-b \pm \sqrt{b^2 - 4ac})/2a$. In order to visualize all the details of a problem he drew pictures of it called flow diagrams. Finally he wrote it up as a list of computer words called a code and fed

it to the computer. All this varied in interest from the fascination of a chess game to the monotony of a hand laundry, but all in all it might be regarded as the washboard era. Hand laundrymen as well as good programmers are hard to find but they do a beautiful wash.

Programming has been expensive too. It has been estimated¹ that code costs from \$2 to \$10 a word in a large computer installation and that the manpower costs as much as the computer rental.

But through the years enlightenment came and from across the sea came subroutines. These were prearrangements of the digits, and once inside the computer they extended the computer's reach across the gap toward the problem. They permitted the computer to absorb and exude decimal numbers and letters and internally to compute square roots, exponentials, logarithms, and many other functions with a minimum of goading.

Subroutines used to be fed into the computer on cards or paper tape along with the codes. Then, with the coming of magnetic tapes, wires, and drums, it was found better to keep the subroutines on magnetic media, readily available to the computer. The codes could still go in on paper media. It now became necessary to provide supervisory or executive routines which read in the subroutines as indicated in the code. If the magnetic media containing all these routines are now included in the black box with the computer, the gap between the problem and the computer is further described.

The programmer now speaks to the black box differently. For example, instead of saying 026000 004046 he says FMP X and the computer is given to understand that it is to multiply the number it is currently holding by X using floating-point operations. Because a list of instructions:

```
LRS 35
FMP X
FAD A+1, J
. . .
```

is not a true code, it is called a pseudocode or symbolic code and the process of converting it to the true machine code has been called a symbolic assembly operation.

A pseudocode may be handled in two different ways after it enters the big black box. If it is immediately converted to true machine code and the necessary subroutines are immediately summoned up, the master routine is spoken of as compiler or assembly program. If the compiler does its work in one sweep through the code, it is called a one-pass compiler. However, if the computer addresses have not actually been assigned by the programmer, it may be necessary for the compiler to pass through the code twice: first to learn what symbolic addresses have been used and assign them actual addresses, and second to convert the pseudocode to true code. This is referred to as a two-pass compiler.

If the pseudocode is left in the computer and is translated to computer code only as the computation progresses, then the executive routine is called an interpretive routine. Of course there are systems that both compile and interpret. The term automatic coding is used wherever computers are made to assist in the preparation of their own true codes.

There are many automatic coding systems in existence, and most of them bear distinctive names such as: SAP, CAGE, ADES, SOAP, FLIP, FLOP, A-1, A-2, B-O, BIOR, SPEEDCODE, BASE 00, PRINT, TRANSLATOR, DUAL, FORTRAN, QUICK, GEPURS. Some of these systems are useful and hence valuable, and some are not. Once a laboratory has invested in a

¹Notes from a series of lectures given by Walter F. Bauer at the University of Michigan Summer Sessions on Digital Computers in 1954, 1955, and 1956

system it may be forced, in order to protect its investment, to continue to use the system although the same investment might have bought a better system. Most of the systems are expensive. A modest system can cost 5 man-years and the total cost of all the above systems would be a staggering figure. Perhaps during this conference we will learn what some systems cost and what they can save. Some people are here as users who wish to learn what to do about getting a system; while others are here as producers who wish to learn what is needed. Perhaps both will get some answers. Perhaps those who have produced a system will tell what ought to be done.

Because the cost of a system is high and talent is scarce, many users of large computers have joined together and formed organizations to develop good automatic coding systems. These are: USE, the Univac Scientific Exchange, PACT consisting of west coast companies and General Electric, and SHARE consisting of IBM 704 users. Although most of the effort in automatic-coding development is straightforward and ingenious, some research has been done. For example, the communication between problem and computer is being examined in the light of information theory and formal logic. The nature of equations and statements is being probed, and in time we can expect the black box to reach clear across the gap to grasp the equations themselves. When equations become the pseudocode the long-sought universal code also will have been found. Universal code designates a pseudocode that is acceptable to more than one type of computer.

The progress of automatic coding might be represented by two graphs. They may prove controversial. If we plot the time required to program a particular problem against the time in years since 1950 it might look like Fig. 2. Subroutine libraries and later compilers and other automatic-coding features have cut down on the amount of time required to get a problem into production.

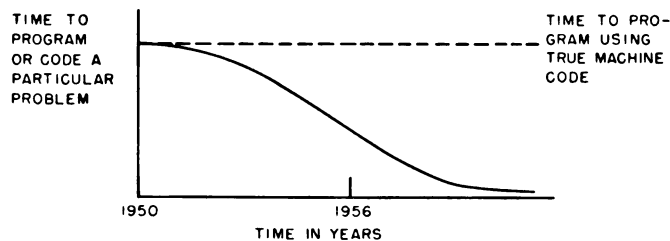


Fig. 2 - The influence of automatic programming on program preparation time

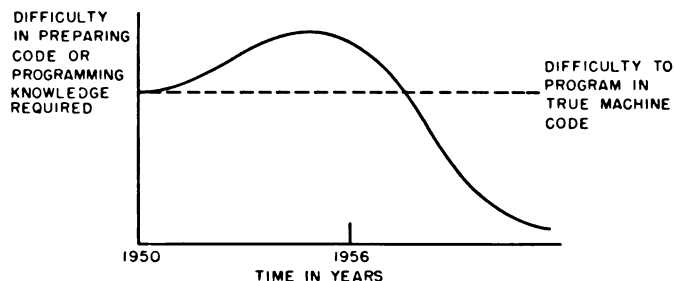


Fig. 3 - The difficulty of program preparation related to advances in programming techniques

However, if we plot the difficulty in preparing a code, or programming knowledge required, for a particular problem against the time in years since 1950 it might look like Fig. 3. With many of the useful automatic coding systems the programmer must first know the computer, its registers, input-output equipment, and other idiosyncrasies. Then he must learn the pseudocode

and the features of the automatic coding system. In many useful systems these are kept simple and similar to the machine code itself. In some systems, where an entirely new pseudocode is used, the overall difficulty of use is greatly increased. However, in time, automatic coding systems should achieve both objectives. They should be very easy to use and greatly shorten program-preparation time. Sending a problem into a computer nowadays is like sending an expedition to Africa to trade with the natives. It has to be complete with the missionaries to translate and possibly convert the natives. If the missionaries speak only French and the native tongue then we must speak French but if the missionaries speak English too then everything is all right.

DEVELOPMENT OF COMMON LANGUAGE AUTOMATIC PROGRAMMING SYSTEMS

Charles E. Thompson

*Hanford Atomic Products Operation
General Electric Company*

COMPUTING AND DATA PROCESSING AT HANFORD

The Procedures and Computing Section has the responsibility of operating and maintaining the central data-processing center and providing mathematical, numerical, and procedural analysis services to the entire Hanford operation. Since June of last year, the data-processing center has operated an IBM 702 system containing 10,000 positions of high-speed electrostatic memory, a 60,000-character drum, nine tape units, a card reader, card punch, and two printers. The machine is being used for a wide variety of applications - everything from payroll and inventory control to reactor design computations and meteorological data reduction.

The demand for 702 services has steadily increased and is rapidly approaching the saturation point. This is illustrated in Fig. 1 in which utilization figures for the month of March have been compiled with machine time distributed by type of application. During the month, the total productive time was 420 hours - an average of 19 hours a day. Of this, 110 hours were used for technical computing and 310 for commercial data processing. To help alleviate the 702 load, the system will be expanded in July when a 20,000-position magnetic-core memory will replace the existing electrostatic memory. In addition, an IBM 650 drum calculator has been ordered with delivery scheduled sometime in August. It is anticipated that the 650 will relieve from 10 to 20 percent of the total 702 load primarily in the area of nonroutine scientific computing.

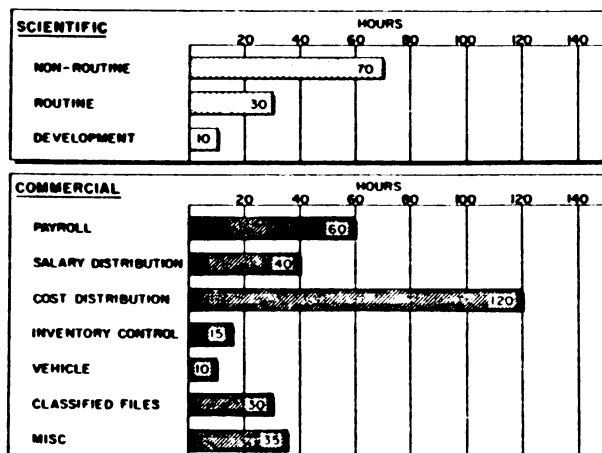


Fig. 1 - 702 utilization during March 1956

Because of the wide variety of work that is done on the 702, our analysis and programming personnel have been organized into application groups or teams. Each team is responsible for the analysis, programming, coding, and debugging of assigned problems. Team responsibility has been made flexible with some groups responsible for a single application, while others are

involved in a variety of related applications. In the last year and a half, these teams have prepared over 700 programs for the 702, involving some 200,000 written instructions.

Figure 2 illustrates the number of different programs required for each of the major areas of application; during March, 153 different programs were processed on the 702 with 74 and 79 in the technical and commercial areas respectively. As would be expected, a large percentage of the programs are of the "one-shot" scientific variety. These normally account for less than one-sixth of the machine time but constitute over one-third of the total programming load.

One final breakdown of machine statistics was made. During March, 793 different jobs were run on the 702, an average of 36 each day. This included 369 debugging attempts, 51 program assemblies, and 373 production runs.

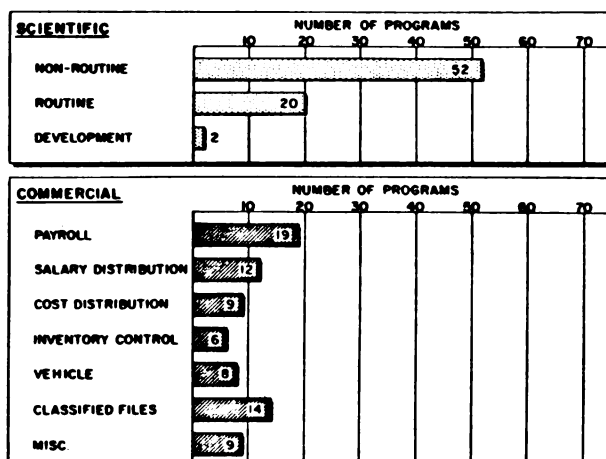


Fig. 2 - Program statistics for March 1956

SCRIPT

The brief description of our work in computing and data processing was presented primarily to help answer two of the questions which enter into decisions with respect to the development of automatic programming systems: first, "Shall we set up an automatic system?" and second, "Shall we prepare our own system or borrow one from another installation?"

Unless the first question is answered in the affirmative, the only alternative would be to write programs in the abstract language of machines. In all but extreme cases, machine language programming is prohibitive. Certainly an installation such as ours with a diversity of applications requires some type of automatic programming system.

The second question - "Should we develop a system of our own or borrow one from someone else?" - was answered soon after the 702 was ordered. Actually we had no choice; we were in the position of having to develop our own system because there were no other 702 systems which satisfied our one basic requirement - compatibility. Originally, we required a system which would provide a common programming language for both scientific and commercial applications. This requirement was fulfilled with the development of SCRIPT. With the ordering of the 650, we began working on a new system called OMNICODE, which will make available a single programming language for the entire range of computer applications on either the 650 or 702.

When we began working on SCRIPT in January of 1955, we had two immediate goals in mind:

1. to provide for the assembly of programs written in the symbolic notation of SCRIPT, and
2. to provide for the compilation of floating-point subroutines.

As it turned out, this was just the beginning. Within a few months SCRIPT was expanded to provide for input-output, tape control, and report-preparation subroutines. In addition, a program-generating section was added to the system which composes instruction patterns from specified parameters. The generated instruction patterns are not classed as subroutines because they are composed directly by SCRIPT, having no predetermined form.

In the symbolic notation adopted for SCRIPT, all memory locations and operation codes are symbolized. Five characters are used for expressing symbolic locations and addresses; 702 operation codes are described by a two-character mnemonic code, and three characters are allotted for pseudo or suboperation codes - suboperations being those which refer to the compiling and generating sections of SCRIPT.

From a written program a deck of program entry cards is prepared where each card corresponds to a written line on the programming sheet. During assembly each program entry is assigned an actual machine location, mnemonic codes are interpreted and translated into their actual equivalents, address assignments are made, and if the program includes suboperation instructions, the required instruction patterns are composed and the appropriate subroutines from the tape library are incorporated in the program. The output of a SCRIPT assembly is a deck of cards containing the actual machine code and a proof listing showing both symbolic and actual program entries.

Throughout the development of SCRIPT every effort was made to keep the coding treatment of suboperation instructions as close as possible to that of 702 instructions so that from the programmer's standpoint there would be no difference between the sixty pseudo operations made available by SCRIPT and the thirty-seven "built-in" 702 commands. Thus, the programmer has complete freedom in interspersing 702 instructions with all classes of subroutine and generating routine instructions; no indicative information is required when changing from one mode of operation to another; and, finally, suboperation instructions can be transferred to and modified in the same manner as is done with 702 instructions.

The manner in which a scientific program might be written in SCRIPT notation is shown in Fig. 3, where values of X and Y are punched in cards in floating-point form. For each X and Y, a corresponding Z is to be computed with X, Y, and Z written on tape No. 4.

The first instruction in the program causes a card to be read into symbolic location 70.01.0. We next compute the cosine of Y and add X. The address of each of these instructions is again 70.01.0 but, since Y is in character positions 13-24 and X in positions 1-12 of the record, increments of 24 and 12 are entered. The result of every floating-point operation is automatically placed in a twelve-position storage location called the pseudoaccumulator. The pseudoaccumulator, which is actually a storage assignment in the subroutine library, has symbolic location A2.90.1. Thus, to square the quantity in the pseudoaccumulator we simply give a multiply instruction with an address A2.90.1. The next instruction computes the logarithm of the partial result and Z is obtained by adding the constant one. Z is then stored in positions 25-36 of 70.01.0 and the completed record is written on tape No. 4. The last instruction is a transfer to the beginning of the program.

$$Z = 1 + \log(X + \cos Y)^2$$

INSTRUCTIONS				
CL	LOCATION	OPERATION	ADDRESS	INCR
	01.01.0	REC	70.01.0	
	02	CNF	70.01.0	024
	03	ADF	70.01.0	012
	04	MPF	A2.90.1	
	05	LNF	A2.90.1	
	06	ADF	80.01.0	
	07	STF	70.01.0	036
	08	WT4	70.01.0	
	09	TR	01.01.0	

STORAGE ASSIGNMENT				
CL	LOCATION	LENGTH	SIGN	VALUE
I	70.01.0	081		
I	80.01.0	F12	+	1000000000+00

RECORD LAYOUT X : 1-12
 Y : 13-24
 Z : 25-36

Fig. 3 - A scientific program written in SCRIPT notation

All of the instructions in this program, with the exception of the 702 transfer, are suboperation instructions with associated subroutines. During assembly, the input-output suboperations REC and WT4 are transformed into eight actual machine instructions which select the input or output unit, read or write the record, test for an end-of-file condition, and interrogate the read-write check indicator. Each of the floating-point instructions, those with an F in the right-most position of the operation code, are transformed into two pseudoinstructions which supply the location of the subroutine and the location of the operand to a floating-point interpretive routine. At the beginning and end of each series of floating-point instructions, the necessary linkage to and from the interpretive routine is

established. Finally, the subroutines which have been referred to are assembled into the program. If this sample program were assembled, the nine written instructions would be expanded into a program containing five hundred and fifty-two actual machine instructions. At the present time, there are thirty-two floating-point subroutines available in the SCRIPT system. In addition, provision has been made for automatic address modification by introducing index registers which closely resemble the built-in variety available on the 650 and 704.

A data-processing program (Fig. 4) is written in SCRIPT notation as follows: A report is to be prepared from a payroll master file with control totals to be accumulated and printed when the entire file has been processed. The first instruction is again an input suboperation which causes a 781-character payroll master record to be read from tape No. 4 into symbolic location 70.01.0. The next instruction is actually a parameter entry for the read instruction and during assembly is incorporated as the end-of-file address in one of the eight generated instructions. The instructions MOV 70.01.0, TO 80.01.0, TO 90.01.0 cause fields in the payroll record to be entered into the report line at 80.01.0 and accumulated in the control total area at 90.01.0. For assembly purposes, SCRIPT requires sets of parameters specifying the location of corresponding fields in the three records and the manner in which each of the fields is to be handled. The programmer records these parameter entries on a Move-To assignment sheet. The first reaction to the Move-To technique is that it requires as much work to prepare the assignment sheet as it would to write out the instruction pattern itself. This is true only in the simplest programs. In most programs there is a significant saving because, once a record has been defined on the assignment sheet, it can be used in any number of Move-To sequences. We have estimated that the Move-To generating feature of SCRIPT has reduced programming and debugging time in most commercial programming by a factor of from five to ten.

Getting back to the example, the instructions at 01.06.0 and 01.07.0, PTL and DE1, cause the assembled report line to be written on tape No. 3. The DE1 instruction is again a parameter entry which specifies the spacing desired on the report, in this case a detail line single spaced, with the tape unit on which the report is to be written entered as the address of the instruction. The PTL or report preparation subroutine has been another time saver. It provides for automatic insertion of headings on each page of the report, form-to-form ejection, internal skipping within a page, printing of totals, and page numbering.

After the report line has been written on tape No. 3 a transfer is made to the beginning of the program and the processing of the next record. When the entire file has been processed, the end-of-file transfer is made to 02.01.0 where the control totals are moved to a print pattern which is printed on the printer. The last instruction is an end-of-job halt. If this program were assembled, it would contain over 150 actual 702 instructions.

We have found the combination of input-output suboperations, along with the Move-To generator and the report preparation subroutines, invaluable in our commercial programming. In fact, they have proven as important in this area of application as have the floating-point subroutines in our scientific programming.

Fig. 4 - A data-processing program written in SCRIPT notation

INSTRUCTIONS			
CL	LOCATION	OPERATION	ADDRESS
01	01.01.0	RT4	70.01.0
02	02	EF	02.01.0
03	03	MOV	70.01.0
04	04	TO	80.01.0
05	05	TO	80.01.0
06	06	PTL	80.01.0
07	07	DE1	TAPE 3
08	08	TR	01.01.0
09	02.01.0	MOV	90.01.0
10	02	TO	91.01.0
11	03	WRP	91.01.0
12	04	HA	99.99-

STORAGE ASSIGNMENT			
CL	LOCATION	LENGTH	DESCRIPTION
1	70.01.0	781	PAYROLL RECORD
1	80.01.0	120	DETAIL PRINT
1	90.01.0	080	CONTROL TOTALS
1	91.01.0	120	FINAL PRINT

MOVE-TO ASSIGNMENT			
CL	FIELD	TO	FROM
3	PR#	008:005042	PRT
3	NAME	008:020040	080
3	NET PAY	35	810082:PRT010:TOTO43:PRT

Our experiences with SCRIPT have been encouraging. The system has done much to ease our training requirements, it has speeded up programming time considerably, it has reduced the computer time required for debugging, and it produces actual programs which utilize memory and computer time more efficiently in most cases than the average "long hand" program. If SCRIPT were compared with a standard symbolic assembly system, it is estimated that the system has saved us 50 man-years in programming and debugging time - a saving of over \$300,000. The total cost for developing SCRIPT was less than \$10,000 - approximately two weeks rental cost for the 702.

OMNICODE

Our need for a common programming language has been satisfied with **SCRIPT**. It has, however, become increasingly apparent that symbolic systems offer, at best, a compromise between the rigid, abstract language of computers and the English and mathematical language in which problems are formulated. A system which would enable programming directly in the literal language of the application would be extremely valuable and would offer a flexibility not possible with a symbolic system. This was our goal when we began developing the successor to **SCRIPT**, which we are calling **OMNICODE**. Originally it was to have been a literal language 702 system to be used for both scientific and commercial programming. However, with the ordering of the 650, we have extended the common language concept to include programming for both the 650 and 702. Throughout the development of **OMNICODE** an attempt has been made to incorporate features which will bridge the gap between applications and machines. This has been accomplished by designing storage layout forms which are prepared in the language of the application, and using this language directly in the preparation of the written program.

Some of the features we are planning to incorporate in the **OMNICODE** system include:

1. functional storage layout forms for the three basic input-output media - cards, tape, and printed report
2. flexible literal assignments of memory locations, operation codes, and operands
3. a functional operation vocabulary designed to satisfy the programmers' needs, ignoring as much as possible the idiosyncrasies of the computers
4. a single-operation vocabulary for both floating- and fixed-decimal arithmetic commands
5. provision for constant and working storage definition directly on the programming form
6. provision for automatic restarts and check points as a part of every assembled program
7. provision for automatic address modification with the number of index registers dependent upon the requirements of a given program
8. automatic decimal control for fixed-decimal programming
9. multiple levels of labeling and addressing to assist in the preparation of large programs which normally required partitioning.

Let us look at the scientific programming example we used to illustrate the use of **SCRIPT** and see how it looks with **OMNICODE** (Fig. 5). Again, given values of *X* and *Y* we wish to compute *Z*. The title "variables" has been assigned to the input record with the fields labeled as *x*, *y*, and *z*. Since each of the fields is in floating-point form, an *F* is entered in the length column opposite each field definition. This is important because the manner in which fields are defined determines whether the computation is carried out in floating- or fixed-decimal arithmetic.

The program reads as follows:

READ the record VARIABLES	ADD the constant 1
Take the COSINE of <i>y</i>	The result IS <i>z</i>
ADD <i>x</i>	WRITE the VARIABLES record
MULTIPLY the result by itself	TRANSFER TO THE START
Compute the LOGARITHM of the previous result	

$Z = 1 + \text{Log}(X + \cos Y)^2$

INSTRUCTIONS			
LINE	LABEL	OPERATION	OPERAND
01	START	READ	VARIABLES
02		COS	Y
03		ADD	X
04		MULT	
05		LOG	
06		ADD	(I)F
07		IS	Z
08		WRITE	VARIABLES
09		TR	START
10			
11			

RECORD LAYOUT			
LINE	LABEL	LEN	DEC
01	VARIABLES		
02	X	F	
03	Y	F	
04	Z	F	
05			
06			

Fig. 5 - A scientific program .
written in OMNICODE

1. If an instruction refers to a record or field which has been defined on a storage layout form, the appropriate label is entered as the operand.
2. If the instruction refers to the result of the previous operation the operand is left blank.
3. If the instruction refers to another instruction, the label of that instruction is entered as the operand. Notice that the only instructions which require labeling are those which are referred to elsewhere in the program, although any of the instructions may be labeled.
4. If the operand entry is one which refers to a constant, the actual value of the constant is entered as the operand. In cases where a constant is required elsewhere in the program, its value can be entered along with a literal title which can then be referred to directly. For example, the constant π might be entered as PI (3.14159). From then on, the constant is referred to by simply entering the letters PI as the operand.

5. The last type of operand, one which was not used in this example, is one which refers to a previously undefined quantity, normally a working area. In this case, the quantity is immediately defined by entering a descriptive label as the operand.

The use of OMNICODE in writing a data-processing program is illustrated in Figs. 6, 7, and 8. This is a simplified data-reduction problem in which a printed listing and a deck of result cards are to be prepared from an input file of sample analyses.

Figure 6 illustrates the layout of the input record which has been labeled "analysis card" and one of the output records described as the "result card." Each of the fields within the records has been assigned a label with the format of each entered under the appropriate heading. For example, the CORRECTION field is described as a five-digit quantity with three positions to the right of the decimal. Notice that the field labels "sample #" and "type" are duplicated in the two records.

The layout of the other output record, the printed listing, is shown in Fig. 7. The report layout form has been designed to provide a visual picture of the final output. Across the top of the form, the desired page and columnar headings for the listing are described. These headings are key-punched directly for input to the OMNICODE assembly. The remainder of the form is used to describe the various types of lines which are contained in the report. For example, a report might consist of a detail line, a minor total line, and a grand total line. In this example, the listing consists of a single type of line which has been given the label "report." The format of the line is described by a series of X's which specify the length of each field and periods which indicate the location of the decimal point. The literal description or label of each field is entered on a vertical line drawn to the right of the field format. For example, the ACTIVITY field, to be listed in positions 97-105, is described as an eight-digit quantity with three decimals.

RECORD LAYOUT			
LINE	LABEL	LEN	DEC
01	ANALYSIS CARD		
02	SAMPLE #	5	
03	TYPE	1	
04	MONTH	2	
05	DAY	2	
06	YEAR	2	
07	REGISTERS	3	
08	LIGHTS	2	
09	TIME	2	
10	CORRECTION	5	3
11			
12	RESULT CARD		
13	SAMPLE #	6	
14	TYPE	1	
15	ACTIVITY	8	3
16			

Fig. 6 - A data-processing program written in OMNICODE - record layout

REPORT LAYOUT

REPORT LAYOUT									
HEADING									
NO. SAMPLE TYPE MO. DATE YEAR RADIO- ACTIVITY									
RECORD									
REPORT									
SAMPLE #									
TYPE									
MONTH									
DAY									
YEAR									
ACTIVITY									

Fig. 7 - A data-processing program written in OMNICODE - report layout

INSTRUCTIONS

LINE	LABEL	OPERATION	OPERAND
01	BEGINNING	READ	ANALYSIS CARD
02		TAKE	REGISTERS
03		MU	(64)
04		ADD	LIGHTS
05		DIV	TIME
06		MULT	CORRECTION
07		IS	ACTIVITY, RESULT CARD
08		IS	ACTIVITY, REPORT
09		MOVE	ANALYSIS CARD
10		TO	RESULT CARD
11		TO	REPORT
12		WRITE	RESULT CARD
13		PRINT	REPORT
14		TR	BEGINNING
15			
16			

$ACTIVITY = \frac{64R+L}{T} \times C$

Fig. 8 - A data-processing program written in OMNICODE - instructions

The program for this example is shown in Fig. 8. After reading an analysis card, the sample activity is to be computed. This requires multiplying the number of registers by 64, adding the number of lights, dividing by the counting time, and multiplying by the correction factor. Since the number of digits and decimals in each of the fields have been specified on a storage layout form, it will be possible for OMNICODE to provide automatic decimal control such that the necessary shift and round instructions are inserted to obtain a result with the desired accuracy. With this feature, the written instructions required to calculate sample ACTIVITY are simply:

TAKE the REGISTERS
 MULTIPLY by 64
 ADD the LIGHTS
 DIVIDE by the counting TIME
 MULTIPLY by the CORRECTION factor

The result IS the sample ACTIVITY which is entered on both the RESULT CARD and REPORT records. Since the activity field is described in each of these records, both the field and record labels are entered as the operands of the two instructions.

The instructions

MOVE the ANALYSIS CARD
 TO the RESULT CARD and
 TO the REPORT

will cause fields within the records having identical labels to be relocated. For example, the sample # in the analysis card will be entered into positions 1-5 of the result card and into positions 1-5 of the report.

The instructions

WRITE the RESULT CARD
 PRINT the REPORT and
 TRANSFER to the BEGINNING of the program

complete the program. The print instruction is similar to the PTL instruction we are using with SCRIPT.

The concept of multiple levels of labeling and addressing is of sufficient importance to warrant additional clarification. The use of record and field labels in describing data has been illustrated in the programming examples. A similar technique is used in the assignment of labels to instructions. Here we use the terms "region" and "step." A region label is used to identify a sequence of instructions. It is normally associated with the first instruction of a sequence and describes the function to be performed by that sequence. Within a region, step labels are used to identify individual instructions. Again, only those instructions referred to by other instructions require labels. In assigning descriptive titles to data and instructions, it is required that all labels within a level be unique. That is, region labels and record labels cannot be duplicated. Similarly, the step labels within a region and the field labels within a record must be unique. It is not required, however, that steps within different regions and fields within different records be unique.

Actually, there is a third level of labeling available. This level, called a "phase," is intended to facilitate the preparation of large programs - programs which normally would be written by more than one person. Each programmer would be assigned a phase of the program, with its associated title, and he would have complete freedom in describing regions, steps, records, and fields within his phase without fear of duplicating labels used in other phases of the program.

It may be of interest to note that OMNICODE programming for the 650 would not be possible without the availability of a large machine such as the 702. Because of the logical complexity of the OMNICODE assembly routine, its large files of data, and the use of alphanumeric source data, assembly of OMNICODE programs for the 650 will be done on the 702. This is not intended as an argument for renting a 702 to do your 650 assemblies, but illustrates the manner in which a larger machine can vastly enhance the flexibility of a smaller machine.

SUMMARY

Our approach to the problem of automatic programming techniques has been dictated to a large extent by the type and volume of data processing which we do. This data processing is highly varied, both with respect to size and complexity, and requires that we use a programming system which minimizes the human effort required to get the job on the machine. This approach is not entirely compatible with the concept of perfect machine utilization, since programs written semiautomatically require machine processing before they can be run. Nonetheless, the economic advantages of the assembly-routine approach have been brought home forcefully time and again. The average SCRIPT assembly requires between 5 and 10 minutes of 702 time, which is an exceedingly small price to pay for the hundreds of man-hours saved in programming applications with SCRIPT. The OMNICODE system will be more comprehensive than SCRIPT, and might require slightly more machine time to assemble programs. But OMNICODE is being developed with just one aim in mind - to reduce the time spent and human errors incurred in placing jobs on a machine. We feel confident that we will again realize substantial economic benefit from this approach, and at the same time take one more step toward reducing the drudgery and enhancing the creativeness inherent in man's relationship with automatic data-processing machines.

PRODUCTION OF LARGE COMPUTER PROGRAMS*

H. D. Benington†

*Lincoln Laboratory
Massachusetts Institute of Technology
Lexington, Massachusetts*

INTRODUCTION

At the 1955 Eastern Joint Computer Conference, Dr. Jay Forrester suggested that the evolution of electronic digital computers might be roughly divided into five-year periods, each period with its paramount significance. To quote Dr. Forrester:

"1945 - 1950 was the period of electronic design. From 1950 - 1955, attention has been focused on the solution of scientific and engineering problems. 1955 - 1960 will encompass the upswing in the commercial data-processing applications. 1960 - 1965 will probably mark the shift of major attention to the use of digital computers as the central elements in real-time control systems."

With respect to this last period, Dr. Forrester continues:

". . . General purpose digital computers, as outlined in [recent news] releases, are to be the nerve centers for tying together the flow of information in our forthcoming new air defense system. This type of control system, we can assume, will develop further into a high-speed automatic control and regulation of future civilian air traffic.

"[Or,] consider the chemical plants and oil refineries. . . . in the last thirty years the automatic controls in an oil refinery have risen . . . to some 15 per cent of the investment in a refinery[or often about]\$15,000,000 worth of automatic controls. I believe we will see digital computers as controllers and monitors of operation in these plants to permit closer control, higher-speed chemical reactions, larger outputs, and a better product."

During the past five years, we have seen developments in automatic programming where the emphasis has paralleled Dr. Forrester's first three periods. We can compare the electronic-design phase with the development of basic programming techniques of translation, compilation, and interpretive routines. Scientific and engineering calculations have been assisted by the PACT and A-2 compiling systems; commercial data processing, by BIOR and B-0 (to name but a few). More important, our colleagues who build computers have come to realize that a computer is not useful until it has been programmed, and that programming is an expensive job which requires both machine assistance and human sympathy.

This paper looks ahead at some programming problems which are likely to arise during Dr. Forrester's 1960 - 1965 period of real-time control applications. At first glance, these are problems which will result from the need for very large, very efficient programs, where one program (consisting of over 100,000 machine instructions) may be used in several machines during periods of months or years. On closer inspection, we realize that these are problems which must be faced whenever the need arises for the systematic preparation and operation of large, integrated programs, whether these programs are used for commercial processing, scientific calculation, or program preparation itself.

*The research in this paper was supported jointly by the Army, Navy, and Air Force under contract with the Massachusetts Institute of Technology.

†Now with RAND Corporation, Lexington, Massachusetts.

During the past several years at the Lincoln Laboratory, several system programs containing over 30,000 machine instructions each have been prepared. These programs are used for data processing and control in real-time systems. Production of these programs is briefly described below, particularly in terms of cost and organization. Four problem areas are stressed:

The first is computer operation. Computer time is at a premium when a large program is being prepared by relatively inexperienced programmers, when the machine and its terminal equipment are being shaken down, and when the machine-program system requires inordinate testing and debugging. The only answer is highly systematic, highly mechanized program preparation and computer operation. A Lincoln Utility System of service routines containing 40,000 instructions has been prepared to ease this problem.

The second problem is program or system reliability. Needless to say, a large program is distressingly prone to all types of design and coding errors, including some very subtle ones. In spite of this tendency, it must be extremely reliable if it is to control effectively a system involving extensive equipment or manpower. This is not only true in a real-time system, but also in commercial applications unless equipment engineers can outvote lawyers. Reliability is also a major factor in the preparation of ambitious automatic programming systems — how many unreliable programs have been produced with supposedly well-tested compilers?

Next, there is the problem of supporting programs. It has been the experience of the Lincoln Laboratory that a system of service programs equal in size to the main system program must be maintained to support preparation, testing, and maintenance of the latter.

Finally, there is the problem of documentation. In the early days of programming, you could call up the programmer if the machine stopped. You seldom modified another man's program — you wrote your own. Although present automatic programming technology has done much to make programs more communicable among programmers, there is a long way to go before we can take an integrated program of 100,000 instructions and make it "public property" for the user, the coder, the tester, the evaluator, and the on-site maintenance programmer. The only answer seems to be the documentation of the system on every level from sales brochures for management to instruction listings for maintenance engineers. Such documentation will require the development of new methods and new "languages"; more significantly, it will require a much more extensive use of the computer to assist in program production, documentation, and maintenance.

At the last ONR symposium on automatic programming held two years ago, the most popular theme was simplifying program input through the use of symbolic inputs, machine compilation and generation, algebraic translation, etc. Very little was said about checkout or debugging, training, or operation. I suspect that for many the past two years have been a period of realizing that automatic programming concepts must go beyond the input process into these other areas.

LARGE PROGRAMS FOR CONTROL AND PROCESSING

Before considering these problems in more detail, consider some rudiments of large systems and large programs.

Figure 1 represents a broad flow chart of a typical control and processing system such as might be used for air-traffic control, industrial-plant control, or commercial applications. The area inside the dashed line represents the control system; the area outside, the environment to be controlled. In general, control consists of a manual and an automatic component. Manual in-out data could use voice phones or radios, teletypes, meters, etc. Typical automatic inputs and outputs might be teletype data or high-bandwidth digital data from or to analog-to-digital converters.

The central control is a high-speed, general-purpose, digital machine which includes in-out terminal equipment and which is controlled itself by the system program. Depending on the degree of system automation, manual control and processing might range anywhere from one half-awake computer operator (who will be awakened by an alarm) to a staff of several hundred

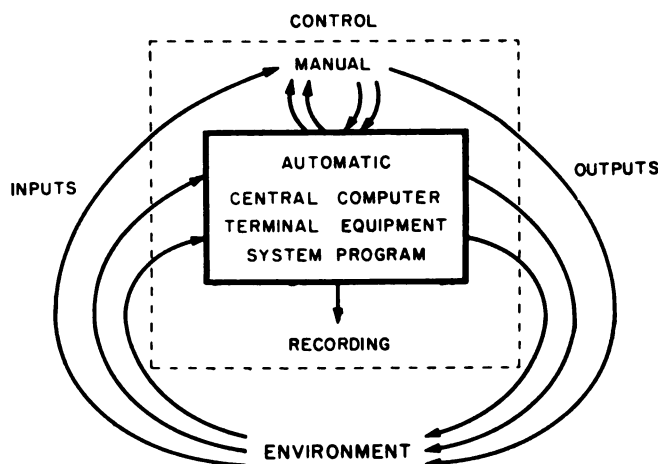


Fig. 1 - Typical control system - In general, a typical control system uses automatic and manual elements. The automatic portion consists of a centralized digital computer, terminal equipment communicating with the environment, and a computer program incorporating system memory and standard operational procedures.

operators and supervisors, each of whom must communicate directly with the computer. The machine can signal the man through indicator lights and alarms, cathode-ray displays, or printed data; the man can respond with digital keyboard inputs or a variety of analog-to-digital devices. Periodically, the computer records data for later analysis of system performance.

From the computer's point of view, then, the system consists of a wide variety of inputs and outputs, each with different data characteristics — peak rate, average rate, reliability, coding, etc. The system program must perform a wide variety of tasks:

1. It must remember the state of environment. Depending on the application, this may require from 100,000 to many billions of bits of information stored on drums, tapes, or photographic plates.
2. It must sample each input either periodically or on demand, translate the data, test for reasonableness (usually in terms of the present state of the environment), and either revise its memory content accordingly or transmit the data for further processing.
3. It must, either periodically or on demand, calculate, monitor, correlate, predict, control, summarize, record, and decide.
4. It must encode and transmit outputs to all terminal devices.
5. Finally, the program must control the frequency and sequence with which it performs each input, output, processing, or bookkeeping task.

In order to give these features some physical meaning, let us attach rough numbers to a typical control problem. Figure 2 shows the organization of a typical 100,000-instruction program which contains 80 component subprograms. In other words, each subfunction requires a logically distinct subprogram containing an average of 1250 instructions. In the figure, each box (e.g., I12) represents a subprogram; they are grouped as follows:

1. There are four major input channels (e.g., punched cards, teletype, audio-bandwidth data link, and manual keyboards) designated by program groups I1 to I4. For each channel, several different types or sources of data are received by the control element. For example, I3 requires seven subprograms, I31 to I37.

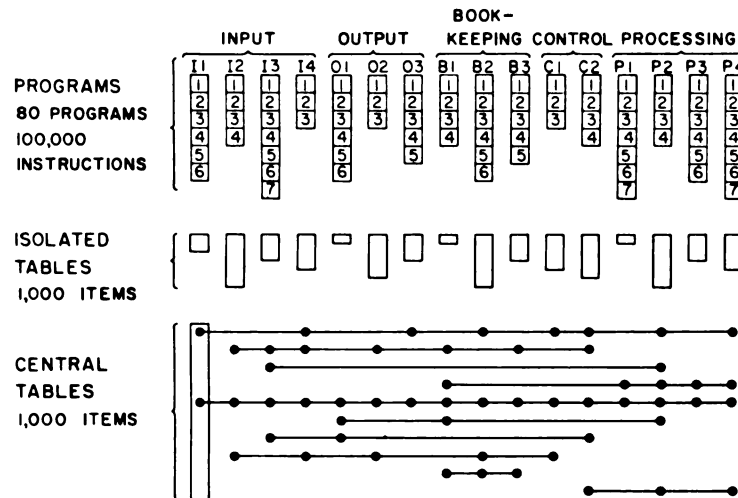


Fig. 2 - Static program organization - A system program of 100,000 instructions is organized into programming groups for input, output, etc. Each group contains several subprograms and requires both isolated and central tables.

2. There are four major processing functions, which require a total of twenty-four component subprograms. In an air-traffic-control application, a typical process might be: First, review all aircraft which are landing at all airports; next, monitor these with respect to air-space assignment and sudden trouble situations; finally, prepare a revised space assignment.
3. A third group of fifteen subprograms are required for program bookkeeping. These programs coordinate communications between all other programs, monitor system load, and prepare summary data for output.
4. The output makeup programs use three channels; for example, cathode-ray display, audio-bandwidth data link, and teletype. Fourteen subprograms are required to scan the system memory and make up properly coded output messages.
5. Finally, seven control subprograms are required to control the timing, sequencing, and operation of all other subprograms.

The 100,000 instructions represent standing operational procedures for the system; they do not change as the system operates. The system memory, which is stored separately in system tables, can be broken down into two blocks: isolated tables which store information required by one program group only (e.g., I2), and central tables which store data shared by two or more program groups. In measuring the complexity of the table structure, the total table memory required by tables is not nearly so important as the number of items. In this sense, an item is defined as one unique type of information. A single item may be represented once in the tables (e.g., "process I42 is being performed"), or the item may be represented one million times (e.g., "customer account number").

In the example given, one thousand items each are required for the isolated and central tables. For ten of the central items, the program groups which set or use the item are shown; for example, the first item is used by I1, I4, O3, B2, C1, C2, P2, and P4. If a thousand such lines were drawn, the dot matrix would measure the communications (and complexity) within the program.

Figure 3 shows how the component subprograms time-share the machine to perform the control and processing functions (only a small portion of the complete program sequence is shown). Each component subprogram requires its isolated tables, pertinent portions of the

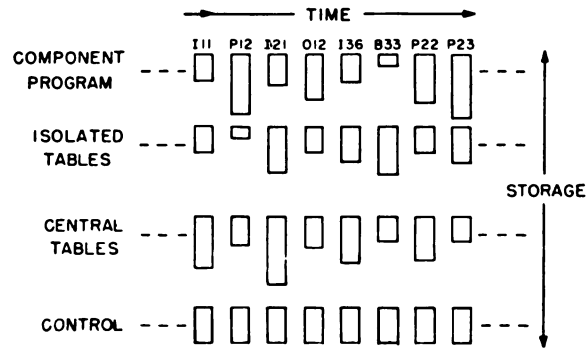


Fig. 3 - Dynamic program operation - Component subprograms (Fig. 2) time-share the control computer. Each component program requires isolated and central tables; a control program, which remains permanently in storage, directs sequence and frequency of operation of component subprograms.

control tables, and certain control subprograms. Eighty programs must time-share the machine. In general, some subprograms will operate unconditionally in a fixed sequence but at different frequencies; other programs will operate on demand.

LARGE-PROGRAM SYSTEMS — CENTRALIZED VERSUS DECENTRALIZED

At this stage, we can consider the effect of program size and integration on the design, testing, and operation of the program.

To date, there have been several "programming systems" of over 50,000 machine instructions prepared for business and scientific applications. For the most part, however, these programs have been what might be called large decentralized programs; that is, the data-processing function has been divided into a dozen or so parts, and the communication between these parts has used blocks of data stored on magnetic tape or punched cards.

Usually, the format and coding (i. e., the structure) of these blocks can be unequivocally defined with relative ease. This considerably simplifies the design problem: after the blocks have been documented, groups of programmers can be assigned to each part with the assurance that little communication between these programmers will be necessary. If the fullest decentralization is desired, the component programs will not share machine storage or machine time. (In some applications, even different machines are used.)

Control of data processing in a decentralized system is primarily manual. Tape reels and programs are changed by computer operators (and even shipped to remote locations). If an unexpected result develops, an engineer or accountant or supervisor can print out intermediate data and decide after the fact what course should be taken. Efficient use of computer time need not be closely monitored, since there are no real-time constraints.

In testing or debugging one part of the system, data produced by other parts is not required until the very last moment that the system is put into operation. (Probably many of the decentralized systems presently in operation still contain many minor errors which are being compensated for daily by users who have become accustomed to these minor idiosyncrasies.)

The important point is that one can write a "large" programming system and still maintain a high degree of decentralization. Like most decentralizations, this course produces a system which contains semantic inconsistencies, ambiguities, and errors; operating inefficiencies result from duplication and wasted motion.

Real-time control systems have presented the first computer application where a very large program is required to perform all assigned functions, and yet where the disadvantages of decentralization cannot be tolerated. Success or failure of the system usually depends on efficient use of computer operating time. Internal control of the real-time program must be highly organized if efficient time and storage allocation are to be achieved, if the many in-out devices are to be adequately sampled, and if automatic decisions are to be made when unusual conditions develop within the program or from the external environment.

The control program must be centralized. This complicates design and coding since communication between component subprograms must have a "high bandwidth." The use of each of the thousands of central table items must be coordinated between a hundred or so component subprograms. Organized, readable specifications for the design and coding phase accomplish part of this task. Even then, only the most thorough testing of the entire program insures that system threads have been carefully worked out, that incompatibilities are discovered, and that all contingencies are accounted for.

PREPARATION OF A SYSTEM PROGRAM

Figure 4 indicates nine phases which are used at the Lincoln Laboratory in preparing a large system program:

First, an operational plan defines broad design requirements for the complete control system consisting of the machine, the operator, and the system program. This plan must be prepared jointly by the computer systems engineers and the eventual user of the system.

From this plan, detailed operational specifications are prepared which precisely define the "transfer function" of the control system. In this representation, the computer, its terminal equipment, and the system program are treated as a black box. On the other hand, this description is sufficiently detailed so that programmers can later prepare the system program using only machine and operational specifications. The operational specifications correspond to the equations which the scientist gives a programmer; numerical analysis has yet to be performed.

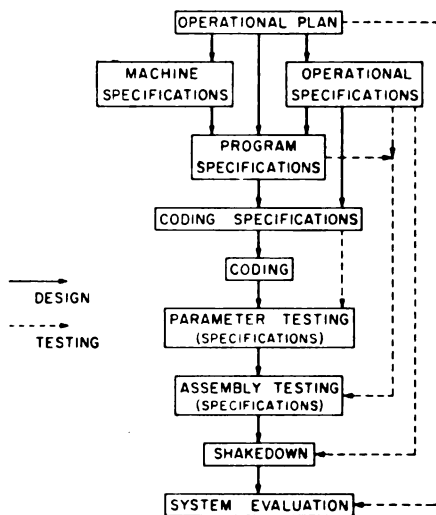


Fig. 4 - Program production — Production of a large system program proceeds from a general operational plan through system evaluation; for example, assembly testing verifies operational and program specifications

Program specifications outline implementation of the operational black box by the system program. These specifications organize the program into component subprograms and tables, indicate main channels of program intracommunication, and specify time-and-storage sharing of the machine by each subprogram. Continuing the above analogy, program specifications correspond to a broad flow chart of the solution.

After the operational and program specifications have been completed, detailed coding specifications are prepared which define the transfer function of each component subprogram in terms of the processing of central and isolated items. From these specifications, it is possible to predict precisely the output of the subprogram for any configuration of input items. The coding specifications also describe all storage tables.

Each component subprogram is coded using the coding specifications. Ideally, this phase would be a simple mechanical translation; actually, detailed coding uncovers inconsistencies which require revisions in the coding specifications (and, occasionally, in the operational specifications).

After coding, each component subprogram is parameter tested on the machine by itself. This testing phase uses an environment which simulates

pertinent portions of the system program. Each test performed during this phase is documented in a set of test specifications which detail the environment that was used and the outputs that were obtained. In the figure, the dashed line indicates that parameter testing is guided by the coding specifications rather than by the coded program; in other words, a programmer must prove that he satisfied his specifications, not that his program will perform as coded. (Actually, test specifications for one subprogram can be prepared in parallel with the coding.)

As parameter testing of component subprograms is completed, the system program is gradually assembled and tested using, first, simulated inputs and, then, live data. For each test which is performed during this period, assembly test specifications are prepared which indicate test inputs and recorded outputs. Assembly testing indicates that a system program satisfies the operational and program specifications.

When the completed program has been assembled, it is tested in its operational environment during shakedown. At the completion of this phase, the program is ready for operation and evaluation.

Figure 5 indicates reasonable production costs which might be expected in preparing a system program of 100,000 instructions. Considering the present technology of program preparation, our experience does not indicate that these are at all overly pessimistic estimates. The estimates shown do not include training of programmers, preparation of ancillary programs, development of control-systems techniques, nor overhead supporting activity. They include only engineering manpower required to produce the system program. Let us assume an overhead factor of 100 percent (for supporting programs, management, etc.), a cost of \$15,000 per engineering man-year (including overhead), and a cost of \$500 per hour of computer time (this is probably low since a control computer contains considerable terminal equipment). Assuming these factors, the cost of producing a 100,000-instruction system program comes to about \$5,500,000 or \$55 per machine instruction. In other words, the time and cost required to prepare a system program are comparable with the time and cost of building the computer itself.

PHASE		ENGINEERING MANPOWER	COMPUTER TIME	PAPER OUTPUT
OPERATIONAL	PLAN	? MAN-YEARS	0 HR	500 PG
OPERATIONAL	SPECS	30	0	2500
PROGRAM	SPECS	10	0	500
CODING	SPECS	30	0	5000
CODING		10	0	3000
PARAMETER	TESTING	20	1000	2000
ASSEMBLY	TESTING	30	2000	1500
SHAKEDOWN		?	?	?
EVALUATION		?	?	?
		130 MAN-YEARS	3000 HR	15,000 PG

MINIMUM PRODUCTION TIME = 18 MONTHS

Fig. 5 - Production cost - Using present techniques, the production cost for a 100,000-instruction program can easily require \$50 per instruction.

THE LINCOLN UTILITY SYSTEM

In order to simplify the preparation and operation of all programs, the Laboratory has prepared a set of service routines called the Lincoln Utility System. This system was designed to assist all programmers in using the machine; its present size - 40,000 machine instructions - is indicative of the importance which is attached to its role. The Lincoln System does not provide automatic-coding facilities in the conventional sense. Compared with systems that have been developed at computing centers where scientific and engineering calculations predominate, the Lincoln System has concentrated more on systematizing computer operation and program debugging, rather than developing automatic translation of programmer language into machine language. Design of the system followed these ground rules:

First, at the Laboratory, most programs are prepared by relatively inexperienced programmers. As many features as possible were included which would help them, yet no features were used which were so complicated that only experienced programmers could use them with facility. Also, programmers do not operate the machine during debugging; they are required to plan and document their operations beforehand.

Next, computer time for parameter testing, assembly testing, and system shakedown is scarce. A large effort has been devoted to systematizing and mechanizing computer operations in order to use minimum computer time.

Third, the Utility System includes several features which assist programmers in communication and documentation problems encountered during the design and testing phases of system program production.

Fourth, the Utility System contains extensive debugging features including facilities for remote, flexible card control of the computer and programs to be tested.

Fifth, programs are prepared in machine language since automatic coding techniques developed to date do not guarantee the efficient programming required for a real-time system. (In retrospect, this ground rule seems very shaky.)

Finally, the Utility System, which is quite large, has not been so centralized that its initial production was delayed or that its revision and improvement are difficult.

With the Utility System, programmers code in floating address using some subroutine requests, particularly for card input and printed outputs. When programs are compiled, they are stored on a magnetic-tape library with their full input structure; that is, the library copy contains program identity, a relative-address binary copy, assigned-memory locations, a floating-address tag table, subroutine requests, etc. Storage in this form has several advantages: First, modifications to a program can be expressed in the floating-address input structure; for recompilation, the compiler does not require a complete program copy. Second, all post mortems during and after program operation are retranslated into input language; programmers do not write programs in symbolic form and receive fixed-address outputs. Third, major modifications in storage addresses and locations can be made to a checked-out program at the time the program is read into the machine since system design parameters are stored in a central communication pool. (See Fig. 6.)

In order to debug programs, a Checker facility is used. This is a service program of 10,000 instructions which allows the program to be tested, or checkee, to be operated either interpretively or noninterpretively under control of a pseudo program of executive instructions. When the checkee is operated in the interpretive mode, the Checker automatically detects loops, arithmetic alarms, illegal in-out sequences, and illegal instructions. It stores a history of program operation including branches, change-registers, and in-out transfers. In the interpretive mode, the checkee cannot cause a machine halt; when alarm conditions are detected, the Checker automatically generates special outputs and moves on to another job. The Checker provides a wide variety of outputs including instruction-by-instruction printouts, dynamic change-register printouts, and alarm printouts. Using the executive instructions, a programmer can set machine registers or memory registers to test values; he can start and stop the checkee at selected locations; he can request different outputs for different regions of the program; he can request

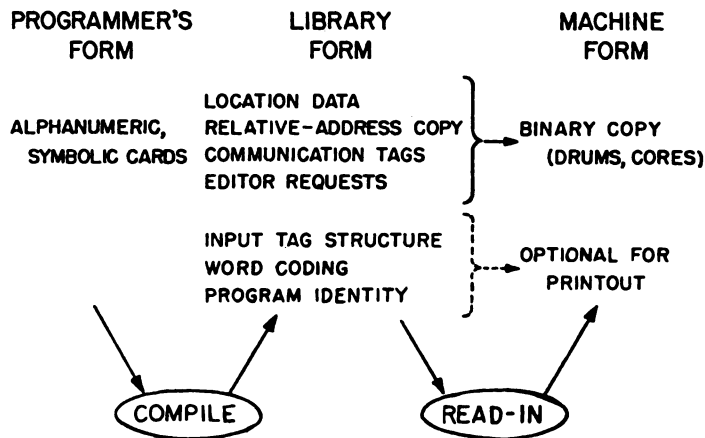


Fig. 6 - Program input process - With the Lincoln Utility System, compiled programs are stored with the programmer's full input structure; at read-in time, the program is finally converted to machine, binary language. Even at this time the symbolic input structure is available to other service routines.

alarm outputs if the checkee transfers control outside a fixed region or if a loop of more than n cycles is performed; he can indicate the use of different executive subprograms depending on the results of checkee operation; he can indicate which portions of his program are to be performed noninterpretively. From a programmer's point of view, the checker is a special-purpose, checkout computer; it is a stored-program machine with highly flexible input, output, and control sections. (See Fig. 7 for a sample executive program.)

All utility programs are controlled by utility control cards. Before a machine run, a deck of binary cards, checker executive cards, etc., is prepared. The operator places the cards in the reader, pushes one button, and the rest of the computer operation is automatic.

A final feature of the utility system is the use of a large communication pool of numerical parameters shared by all programmers. Each programmer can specify that constants or addresses in his program should be taken from the pool. Numbers in this pool are expressed symbolically by the programmer in both his coding specifications and his coded copy; the machine supplies proper numerical values at read-in time. These values may be unknown to the programmer and even changed from day to day. For example, communication tags are used for extracting information (usually table items) which is packed into a full word. The programmer need not know the exact location of the word in memory, nor the position of the information bits within the word. Communication tags are even used to indicate the location in memory of the program itself. A program-design group assigns specific numerical values to the tag pool from day to day, in some cases long after component subprograms have been debugged. Since numerical values are assigned only when the program is read in the machine, it is possible for system designers to move programs and tables within drum and core memory merely by changing constants in this pool. Only one central document needs to be revised and minimum testing on the computer is required.

Figure 8 indicates the allocation of the 40,000 instructions in the utility system.

TESTING

It is debatable whether a program of 100,000 instructions can ever be thoroughly tested; that is, whether the program can be shown to satisfy its specifications under all operating conditions. Considering the size and complexity of a system program, it is certain that the program will never be subjected to all possible input conditions during its lifetime. For this reason, one must accept the fact that testing will be sampling only.

CHECKER CARDS/DELAYED

```

01 NI 11A 11R
02 AL 07
03 LP 25
04 LR 12 13
05 TR 12 13
06 BG 12A 13Z+6
07 LP 4
08 LR 14 15 16
09 BG 14A 16L+5
10 CC
11 QT

```

Fig. 7 - Sample executive program — The Lincoln Checker is controlled by pseudoinstructions. The executive program shown indicates regions of the checkee to be performed noninterpretively (01 NI), alternate executive instructions in case of checkee alarm (02 AL), maximum length loops (03 LP), legal regions of checkee operation (04 LR), checkee output mode (05 TR), etc.

<u>PROGRAM</u>	<u>LENGTH</u>
COMPILER	10 500
READ-IN	1 300
LIBRARY MERGE-OUTPUT	4 700
CHECKER	7 500
MASTER TAPE LOAD	2 000
IN-OUT EDITORS	2 400
COMMUNICATION POOL	4 100
UTILITY CONTROL	3 000
NUMERIC SUBROUTINES	1 000
MISCELLANEOUS	4 000
	<u>40.500</u>

Fig. 8 - Utility system - The Lincoln Utility System requires over 40,000 instructions as indicated

On the other hand, many sad experiences have shown that the program testing effort is seldom adequate. When the program is delivered for operation, its performance must be highly reliable since the control system is a critical part of a much larger environment of men and machines. One error per 100,000 operations of the entire program can easily be intolerable.

As a result of facing this problem for some time at the Laboratory, the following principles have evolved which govern our testing:

First, parameter testing (i. e., testing of individual component subprograms in a simulated environment) cannot be too thorough. This phase must discover all errors internal to the program and its individual coding specifications. Even if parameter testing were perfect (which it never is!), many errors in system design would remain to be discovered during subsequent assembly testing.

Second, initial assembly testing should be performed using completely simulated inputs. There are several reasons: First, only in this way can all test inputs be carefully controlled and all tests be reproducible. Second, when errors are discovered with a new program using live inputs, there will always be a question whether the program or the machine is at fault. Integration of the system program with terminal equipment should not be attempted until the assembled program has been well tested.

A third principle is that the testing facility used during the assembly test phase must contain extensive, flexible facilities for recording both system outputs and intermediate outputs (i. e., subprogram intercommunications). Without this facility, rapid and reliable diagnosis of system errors is impossible. After a test has been conducted and errors found, it should be possible to correct the error before the program is put on the machine again.

The need for comprehensive simulated inputs and recorded outputs can be satisfied only if the basic design of the system program includes an instrumentation facility. In the same way that marginal-checking equipment has become an integral part of some large computers, test instrumentation should be considered a permanent facility in a large program.

Figure 9 illustrates the role of test instrumentation in a system program. Each of the live inputs can be individually simulated; this allows simultaneous testing with both live and simulated data. In addition, the input instrumentation allows easy setting of initial conditions for system memory; this feature is performed by a special-purpose translation program which converts alphanumeric card data into system tables. The output instrumentation "probes" both internal data (for diagnosis) and external data (for simpler verification).

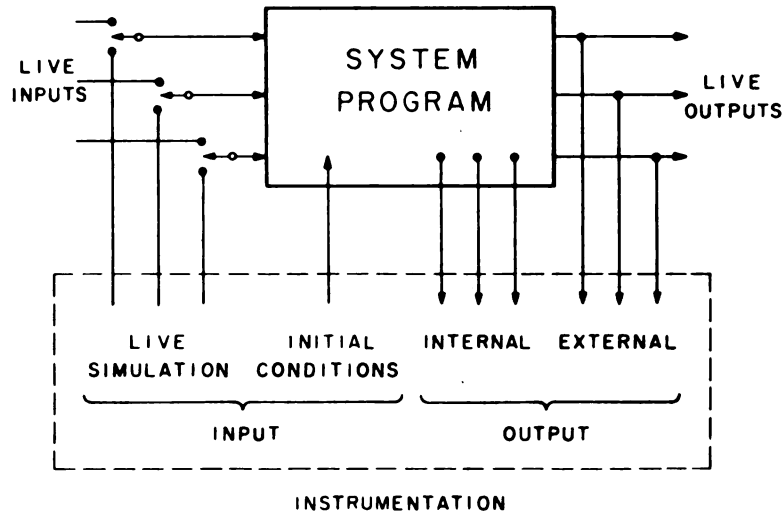


Fig. 9 - Test instrumentation - Proper testing of a control system requires an automatic facility for simulating inputs and monitoring outputs. With this facility, extensive testing can be performed and outputs produced for either diagnosis of system errors or verification of proper system performance.

One final principle should govern system-program testing: All successful parameter and assembly tests must be reproducible throughout the life of the system program. These tests must be documented in test specifications which detail the reasons for the tests, required inputs, operating procedures, and expected outputs.

The original reason for this requirement stemmed from the problem of revising the program once it was operational. The slightest modification to a program can be successful under limited testing conditions and yet still cause critical errors for other operations. Since it is desirable to retest the program thoroughly after each modification, a large battery of test inputs must be available. We have discovered two other incidental advantages of detailed test documentation: First, a programmer's tests tend to be more organized and more exhaustive if he must document them. Second, if machine versus program reliability is ever questioned, then retesting is possible. If a known program and a known test fail, the machine is at fault.

SUPPORTING PROGRAMS

The utility and test-instrumentation programs discussed above are only part of the complete set of supporting programs. In addition, special programs, which assist preparation of the system program, are used to generate routine data blocks, perform special translation of alphanumeric data into parameter tables, assemble program-sequence and timing parameters, etc.

Operational instrumentation programs are used during system shakedown and evaluation. They contain simulation and recording facilities which are far more realistic and operationally oriented than the test instrumentation. System recorded data is analyzed with a battery of data-reduction programs (Fig. 10).

SYSTEM PROGRAM	100,000 INSTRUCTIONS
UTILITY PROGRAMS	40,000
SPECIAL PROGRAMS	10,000
TEST INSTRUMENTATION	20,000
OPERATIONAL INSTRUMENTATION	30,000
	200,000 INSTRUCTIONS

Fig. 10 - Production of a system program - Supporting programs whose total size equals the system program may be required to simplify production and testing of the system program

DOCUMENTATION - DESIGN AND REVISION

As indicated above, documentation of the system program is an immense, expensive job. The output will run to tens of thousands of pages of specifications, charts, and listings. At the Laboratory, these presently include:

- operational specifications
- program specifications
- coding specifications
- detailed flow charts
- coded-program listings
- parameter test specifications
- assembly test specifications
- system operating manuals
- program operating manuals.

The need for this battery of documents is obvious. The system and its program must be learned and used by management, operational-design engineers, system-operating personnel, training personnel, program-design engineers, programmers, program-test engineers, evaluation personnel, and, if more than one system is maintained, by on-site maintenance programmers. Each of these users has very different needs.

Consider the problem of revising the system once the program is operational in the field. A minor change in the operational specifications is proposed. First, the cost and effects of this change must be evaluated in terms of the program, the operators, and, often, the machine. In order to make the change, several hundred revisions may be required in the specifications. If the change is approved, these documents must be changed, operating manuals must be revised, and the program modified and thoroughly tested. The wave of changes must be coordinated smoothly.

Digital computers are often sold to management on the basis of their programmed flexibility. We have said: "If your doctrine or procedure changes, no messy, expensive, time-consuming equipment changes will be required." In reality, this is not true today. The cost of the documentation mentioned above is only a symptom of the design coordination problem in large systems.

How can we reduce this cost? Obviously, as we have done already, by more extensive use of the computer. (At the Laboratory, we have partially gone in this direction through the use of punched cards for storing all central design data. Decks are easily revised, fed into the system program or listed for the user.) We must systematize design, production, and documentation both in the small and in the large. By "in the small," I mean what is already being

done in autoprogramming. However, instead of an algebraic translator we need a unified "bookkeeping - logical - processing - algebraic translator." Before we get this, we will surely need much more research in coding languages and representations. Eventually, programming should become a two-way conversation between the imprecise human language and the precise, if unimaginative, machine. The programmer will say: "Do this," and the machine will answer: "OK, but what happens if . . ." The smallest gain of such a system would be the elimination of the coding, parameter testing, and parameter test specification phases. Unfortunately, these phases represent only one quarter of the system cost.

Documentation "in the large" poses a bigger challenge.

1. What integrated set of documents are required to design and describe a large system?
2. What "language" should these documents use?
3. How should they be cross-referenced?
4. Can we eventually store them on magnetic tape and let the computer analyze, print, and code?

SUMMARY

The techniques which have been developed for automatic programming over the past five years have mostly aimed at simplifying that part of programming which, at first glance, seems toughest - program input, or conversion from programmer language to machine code. As a result of progress in this area (and a growing number of experienced programmers), we find that large programs can now be produced; unfortunately, they are difficult to test and publicize. If the newest very high speed, large-memory computers are to be fully utilized, we must develop automatic programming procedures so that they allow cheap production of highly reliable, easily revised, well-documented system programs.

SHARE - A STUDY IN THE REDUCTION OF REDUNDANT PROGRAMMING EFFORT THROUGH THE PROMOTION OF INTER-INSTALLATION COMMUNICATION

Fletcher Jones

*North American Aviation, Inc.
Los Angeles, California*

The papers presented during this first day of the symposium are largely concerned with the economics of computer usage. One need not long consider the problems involved in this area before he encounters two major impediments to the economical use of computers: the lack of ability or inclination on the part of computer users to communicate efficiently with other installations using like equipment, and a consequent wasteful duplication of effort. A third, less obvious, fault is the relative inefficiency in machine design due to the lack of collective, authoritative feedback from the customer to the manufacturer. Principally because of the "language barrier" and the almost paradoxical lack of common interest, it is rare that many users of similar computers collaborate to offer the manufacturer suggestions for computer design modifications. We, of the organization known as SHARE, feel that we have, to a large extent, successfully attacked these problems. As a result, we believe that we are in a position to help other computer users interested in avoiding the aforementioned areas of difficulty by offering them the benefit of our experiences.

THE NATURE OF SHARE

SHARE is a voluntary, informal organization of the users of the IBM Type 704 electronic data-processing machines. It is devoted to

1. the standardization of machine language and certain machine practices,
2. the elimination of redundant effort expended in connection with the use of the computer,
3. the promotion of inter-installation communication, and
4. the development of a meaningful stream of information between the user and the manufacturer.

It is presently composed of forty-seven installations, including all of the announced prospective users of the 704.

HISTORY

SHARE had its genesis on the west coast. Upon getting thoroughly into preparations for use of the 704, three installations in the Los Angeles area began to have informal discussions concerning their individual plans. Having been pleasantly surprised by the successful cooperative effort for the design and coding of PACT I, a favorable climate existed for a similar joint activity in connection with program development for the 704. It was in this atmosphere, then, that the Rand Corporation of Santa Monica, Lockheed Aircraft of Burbank, and North American Aviation of Los Angeles began to consider seriously standardization and the sharing of the tremendous programming burden with which they were faced.

A fortunate circumstance at this time was the 704 seminar held by IBM in Los Angeles during the week of 8 August 1955. This meeting brought together representatives of seven western installations, and the idea of standardization was discussed among them. The mutual respect which the participants in these discussions had for the competence of the others soon produced the realization that an isolationist attitude no longer existed. The cogency of the standardization and work-sharing concept was such that all professed themselves as being quite willing to accept the ideas of others, even to the extent of obsoleting work already done in the individual installations. It was unanimously agreed by this group that all prospective 704 installations in the country should be encouraged to join the cooperative effort.

Since it was known that several installations had already begun extensive programming projects and that 704 delivery dates, for some, were rapidly approaching, extreme haste was indicated. Accordingly, the initial meeting of SHARE was called for the week of 22 August 1955. In spite of such short notice, almost all of the known prospective installations responded with enthusiasm, and attendance at the first meeting numbered seventeen installations represented out of nineteen contacted. It is a testimonial to the recognized gravity of the 704 preparations situation that this many people, given so little assurance of success, were willing to devote a full week's time to the exploration of the possibilities of cooperating.

MEETINGS

The first meeting of SHARE convened at the Rand Corporation, Santa Monica on 22 August 1955. From the beginning, this meeting was possessed of an aura of intense interest and whole-hearted cooperation. The enthusiasm with which the SHARE philosophy was received was of a degree far greater than even the staunchest supporter had expected. What had begun as a hope on the part of a few that several installations would cooperate had produced, from every represented installation, an "agreement to agree." There was a common feeling among the attendees that the success of the SHARE venture was assured.

It is important to note that a policy developed during the first meeting, one which has governed the activities of SHARE throughout its history, is that, except in those cases where unanimous agreement is easily gained, no standard that is not vital to the avowed purposes of the organization will be established. This fundamental rule, tacitly accepted, has no doubt saved SHARE from the time-consuming and emotion-arousing discussions which have been the common downfall of other organizations.

A series of solid accomplishments resulted from the efforts of the participants in the first meeting. The three necessary tools of communication — assembly program, mnemonic operation code, and card format — were standardized, and a distribution system, the lifeline of any organization, was established. The activities of this group also resulted in standardization on a 704 print wheel format, utility program conventions, and such other items as the description of the location of the binary point. The program writeup form was fixed and volunteers accepted programming assignments such as elementary functions, input-output routines, double precision and matrix abstractions, and differential equation subprograms.

All of the decisions made during the first meeting did not come easily. As was expected, many different views were expressed on each subject and the resolution of the various opinions sometimes came only after hours of discussion. The basic SHARE policy that dissenters should be won over to unanimity by argument and not voted down preserved the spirit of cooperation as compromise replaced strong feelings.

Noticeably, the preparation of bylaws, a charter, and the acceptance of rules of order are missing from the list of accomplishments of the first meeting. These features, usually felt to be necessary for the efficient operation of a sizeable organization, have never been incorporated into the SHARE structure for two reasons: It is felt that a greater degree of flexibility, resulting in more accomplishments, can be attained without the outward flourishes of parliamentary procedure and a rigid set of bylaws; and, since the success of SHARE is dependent entirely upon the willful cooperation of the participants, such devices are considered unnecessary. Clearly, even the most rigid charter could not force the will of the body on an individual installation. The officers of SHARE have "played it by ear" in handling situations that seemed to require

formal procedural action, and, although the time may arrive when larger attendance at meetings indicates the necessity of the adoption of standard procedures, the plan of informality has thus far been of great benefit to SHARE.

The second meeting of SHARE convened in Philadelphia on 12-13 September 1955. This session, held in conjunction with the Eastern Joint Computer Conference, saw the revision of certain decisions made at the first meeting and the consolidation of conventions and agreements into a definite plan for the future. Cooperative programming began to come to the fore as basic policy issues were decided, and the mechanization of the programming effort was completed. It is notable that, of all the thirty-seven programming commitments made during the first meeting, only two were not completed on schedule, while over twenty other programs for which no assignment was made were submitted to the membership. These programming assignments had not been machine-tested, since no 704 had been delivered before that time, but the submission of detailed program specifications made it clear that completion of this work was merely a matter of having the necessary tools. SHARE, within the space of three weeks, had begun to bear tangible fruit.

Subsequent meetings of SHARE in Boston, San Francisco, and Chicago saw the membership grow to the present forty-seven installations, representing an investment in machine rental of some 33 million dollars annually, and the attendance at meetings increased to over a hundred participants. The ground swell of enthusiasm for the movement brought in members from Canada, England, and France.

The accomplishments during and between these meetings have been very gratifying. Some three hundred machine-checked 704 programs have been distributed to the membership. There has been developed a SHARE Reference Manual, which includes such information as a list of all SHARE standards, a bibliography of all 704 programs distributed through SHARE channels, and the machine configuration of each member installation. SHARE has submitted to IBM several suggestions for changes to the 704. This has resulted in several small changes to the machine and one imminent change which will make the 704 conceptually different. There have been subcommittees appointed for action on such projects as education (both of 704 personnel and potential computer users), mathematics, and machine reliability. These groups have made a great deal of progress and are expected to submit shortly to SHARE solutions to troublesome problems.

EFFECTS

Though the full potential of SHARE has not as yet been realized, there are now many beneficial results to recommend the organization as a significant advance in the right direction. It is expected that, in the near future, giant strides will be made toward complete exploitation of the 704.

Probably the greatest effect created by SHARE lies in the area of communication between users. Much-needed distribution systems which simplify the transmission of ideas have been established. The SHARE Manual is a highly authoritative source of information relevant both to 704 programming in the SHARE system and to the members of SHARE themselves. Due to the standardization of the essentials in computer language, technical communication is made far easier -- a program originated at one installation can be read and understood directly by a programmer in another company, instead of requiring a translation stage. Not to be denied, either, are the much underrated but, many feel, very important benefits arising from the personal acquaintanceships developed through SHARE. Many people, who might otherwise never have felt the need to communicate with other 704 users, have been in contact with these in meetings and by mail. The result of these associations has been enlightenment in all quarters and the development of strong inter-installation relationships between groups smaller than SHARE itself, which, in turn, has resulted in much cooperation at the two and three installation level. The content of discussions during the meetings, as well as that of the bar talk during the "evening sessions," has been extremely valuable, for here the members have been able to gather statistics, data, or answers that might otherwise have been available only through exhaustive and expensive effort.

Secondly, SHARE has had a profound effect in the area of 704 programming. Over three hundred checked-out 704 programs have been distributed to the membership, with surprisingly little duplication of effort. Bearing in mind the number of SHARE members, it is almost difficult to believe that only three general printing routines have been written, only one matrix abstraction, and only five square-root subroutines. Needless to say, without SHARE, there now would be at least twenty-five of each among the 704 installations. Clearly, cooperative development of this program library and the consequent obviation of redundant effort has enabled programmers to apply their talents to more elegant utility programs and a much larger volume of applied problems. The lack of urgency in preparing for the 704, resulting from the use of programs written in the cooperative system, has occasioned the development of more sophisticated methods and programs than would have been possible otherwise. We believe that we have not as yet scratched the surface in programming, since only approximately twenty 704's have been delivered to date and SHARE has thus far concerned itself primarily with the development of basic utility programs. Forthcoming in the near future are more powerful utility routines, including compilers, higher powered mathematical routines, and the pooling of experience and knowledge to produce sophisticated diagnostic systems. I have heard estimates which indicate that, by a year from now, one thousand 704 programs will have been distributed through SHARE. This is perhaps optimistic but it illustrates the almost unlimited promise held by the future of SHARE.

A third effect of SHARE is that there has evolved a strong line of communication from the customer to the manufacturer. This is important from the viewpoint of the customer because, through his association with SHARE, he is able to get information concerning design changes, new products, delivery schedules, etc., much faster than ever before. It is extremely valuable to the manufacturer because, through the direct contact of the customer and the manufacturer's high-level design personnel, authoritative information is readily available on such subjects as machine reliability, layout problems, and suggested design modifications. SHARE has been able to provide IBM with collectively considered requests for changes to the 704 and its associated equipment. Through our experience of having obtained excellent results from these suggestions, we are certain that IBM has listened more attentively to SHARE requests than if the same requests were made by a few random users. Although SHARE has decided to concern itself strictly with the 704 and thus has excluded future machines from consideration, the presence of so much authoritative customer opinion in one place is bound to have its effect on the design of the computers of the future. This is obviously fertile ground for the pursuit of both technical and sales research surveys with the expectation of obtaining highly meaningful information.

There are several important effects of SHARE which are byproducts of the original intent of the organization. Primary in this category, as we of the computing field are concerned, is the advancement of the state of the art. It has already been illustrated that, with less time devoted to grinding out the necessary utility routines, more effort can be devoted to the improvement of both mathematical and programming methods. Also, fewer people are required to maintain a computing installation efficiently. Indeed, with the already sizable library of SHARE programs, latecomers are finding that very little manpower need be devoted to readying the machine for useful application. This is a particularly desirable result of SHARE, in view of the critical shortage of experienced computer personnel. In addition to these byproducts, there is a more general advantage gained in that national defense is enhanced through the efficient use of computers, almost all of which are engaged in "essential industry," and of much needed scientific manpower.

THE FUTURE

In addition to the continued development of what seems to be an unlimited potential in the area of programming achievements, SHARE has embarked on several long-range projects which should greatly benefit its members as well as computing in general. Potent ability has been marshaled and channeled into the field of mathematical research. Efforts are being made to educate students in colleges and high schools on the existence and usefulness of computers and profitable careers in the computing field. Although SHARE has thus far conscientiously avoided the discussion and distribution of applied programs because proprietary and government classified standards might be violated, it is conceivable that in the future the exchange of general applied programs will be accomplished, thus further reducing the amount of redundant effort in scientific and engineering computer use. The most profound accomplishments, however, await fulfillment in the direction of programming. Here we may expect to see vast advancements in sophistication which will greatly magnify the usefulness and versatility of the 704 and the competence of the personnel who support it.

CONCLUSIONS

We of SHARE believe that, if the trend toward complexity in the use of computers continues, cooperative organizations of computer installations will, through necessity, play a major role in the exploitation of computing equipment. We have already witnessed, in the past year or two, the phenomenon of a breakdown throughout the computing field of the attitude of splendid isolationism, resulting in several notable cooperative achievements. Specifically, in the wake of SHARE have arisen two other large user's organizations, both of which are experiencing success. The movement is steadily growing.

Those who are interested in reducing their programming burden and gaining the other advantages described here would do well either to generate or to be prepared for the call for inter-installation organization and, once organized, to pursue with enthusiasm the objectives of their group. We of SHARE are confident that they will find, through cooperative effort, a higher level of achievement and economy in the use of computers.

ADVANCED PROGRAMMING TECHNIQUES WITH SMALLER COMPUTERS

J. W. Carr III and B. Arden*

*University of Michigan
Ann Arbor, Michigan*

Perhaps the title of this paper has a more general tone than is warranted by the context of what will be covered. The discussion that follows is concerned specifically with the University of Michigan's experience in operating a small computer, the IBM Type 650, in an open-shop computer laboratory. Even from this limited experience it is possible to make certain generalizations in regard to the problems of such installations. The subject can be covered in three sections: a short history of the 650 computer laboratory, a description of the courses being offered in computer methods, and an indication of desirable future improvements.

If mottoes were desirable for computer laboratories, an excellent one would be: Organize, Standardize, Plagiarize. In the relatively short time that the computer has been in operation these three goals have been pursued — with perhaps the emphasis placed in the reverse order. On 9 March of this year the University received the Type 650 which was intended to be used as a laboratory instrument for certain computer courses and as an open-shop computing center for the University's graduate students and staff. The laboratory staff is small — four half-time people — but fortunately all had had some experience with a large computer, MIDAC, which has been in operation at the University's Engineering Research Institute for three and one-half years. Initially, before the computer was installed, a search was made for an interpretive routine that could be used as a teaching tool — as a first code for novice programmers. MITILAC, an interpretive routine developed by the M.I.T. Instrumentation Laboratory, was selected for this purpose. MITILAC is a floating-point routine using a mnemonic pseudocode and a three-address instruction. The importance of these two characteristics from the point of view of teaching cannot be overemphasized. Such refinements in MITILAC as a differential equation solution subroutine (coded as DIFEQ) have had good acceptance. An ordinary differential equation is computed at one point as a set of simultaneous first-order differential equations, the interval of integration specified, and then the differential equation operation initiates a subroutine which solves the system by the Runge-Kutta method.

Secondly, since efficient use of the 650 demands attention to minimum access coding, a search was made to find the routine for converting a given program to an optimized one. IBM's SOAP (Symbolic Optimal Assembly Program) seemed to more than answer the requirements, since it permitted the original program to be written in symbolic alphabetic form and then converted into optimized straight 650 language by the computer.

The decision was made to standardize on these two programming techniques as the methods to be taught to the staff and students. Accordingly, manuals were printed and used as the primary text in teaching 650 programming. A third important step in standardization was to agree on several standard input-output formats so that the troublesome, yet relatively unimportant for scientific computation, subject of board wiring could be largely ignored. The remaining areas of standardization were subroutine format and utility and checking routines.

*Paper presented by B. Arden

For the latter a changed-word post mortem and a trace were programmed; the primary consideration in both instances was ease of use.

The open-shop operation was intended to be run on a do-it-yourself basis. In order for a student or staff member to do his own programming, punching, editing, and machine operating it was necessary that he receive some type of formal education in the computer. The education was provided by three sources: A relatively few people attended courses offered by IBM. The majority of the users attended one of several twelve-hour courses on SOAP and MITILAC given by the laboratory staff. Enrolled students, both graduate and undergraduate, took credit courses in either computer methods or numerical analysis which used the machine as a laboratory facility. As an indication of the acceptance of the computer, the first twelve-hour course had sixty enrollees and the second, offered several weeks later, a hundred and ten; there were about sixty students in the courses offered for credit. The latter courses, offered in the Mathematics Department, have two hours of lecture per week devoted to the logical design and components of computers. The two-hour lab is aimed at giving the student experience from the start in all phases of computer operation – programming, punching, operating the computer. The problems assigned were taken from elementary algebra, statistics, differential equations. In the course of a semester, students are expected to program fourteen problems which range in difficulty from simple area calculations to matrix operations and differential equations. One section was formed in which data processing was emphasized. As mentioned before, the students were initially taught to use MITILAC, were later taught the 650 code, and finally the assembly routine SOAP. This seems to be a successful order in which to teach programming although it does present some difficulties when the students are checking problems programmed in a pseudocode when they do not understand the basic machine code. MITILAC has built into it a very desirable feature (in fact a necessary feature for this purpose) – a trace routine whose output is consistent with the pseudocode used.

At Purdue University, where there has been in operation for some time an open-shop computer laboratory using the Datatron computer, there is a slightly different attitude in regard to programmer education. They similarly offer short courses in programming and credit courses in computers. For the latter courses, however, there is no emphasis placed on using the computer in laboratory sessions or actually programming problems. A strong argument can be made that the additional theory thus covered will be more valuable than coding skill on a particular machine. Apropos the learning of computing techniques, it is amazing how many people without computer experience are convinced that they can immediately operate the computer at a time when no supervision or assistance is available. It became necessary as an administrative measure to require that all people who desired to operate the computer after hours take a test that demonstrates their knowledge of the console and the most frequent pitfalls of checking. The test is simply a small program which has been bugged and must be debugged and a post-mortem obtained. Strictly speaking not many people have passed the test, but the process of trying has proven to be a good teaching medium for learning checking and the use of utility routines.

Before mentioning some of the problems of small computer installations, it may be of interest to indicate the utilization of the 650 computer recently installed at the University of Michigan:

15-31 March	101 hours
April	355 hours
May	302 hours.

About a third of the time was consumed by students enrolled in courses checking and running laboratory problems. It is important to observe that even in the first two-week period a number of production runs were completed by staff members who had no previous computer experience – such problems as the computation of solar lines, intercorrelation of personality variables, studies on blast load response of tall buildings, etc. These problems were all coded in MITILAC and demonstrate the usefulness of an easily learned interpretive routine.

The advent of production models of small computers presents a possibility, which has not been successfully exploited, of wide spread standardization and easy interchange of programs.

The feature of variable input and output format, i.e., board wiring, in the 650 has prevented an easy interchange of programs. What is needed is either several built-in standard input-output formats which may be selected by the program or for the various users to agree on a standardized format for general, frequently used programs.

In regard to standardization, it would be desirable to have a single code which might be used as input for compilers on various machines. In particular at a large university there are frequently several different groups who will have time available to them on computers not located on the campus. With such a code the training of such groups could be handled by the computer group on the campus. Such a common code seems hardly feasible for machines that differ widely in storage capacity but within the class of magnetic-drum computers this is possible. The initial steps have been taken among university users of the type 650 to cooperate on the formation of such a code. The Purdue University computing group, under the sponsorship of a group of Datatron users, has nearly completed a compiler for use with the Datatron with magnetic-tape units.

The compulsion to cooperate in standardizing formats, compilers, subroutines, etc., seems to vary directly as the cost of the computer involved. The need exists, however, for small computers. In particular, universities operating with small staffs as an open-shop would profit from this cooperation. The large number of general programs obtained as byproducts of teaching computer courses will have only a very limited use unless they can be used on computers other than the laboratory machine for which they were written.

SESSION II - 14 JUNE 1956

Chairman: Albert E. Smith

***Bureau of Ships
Washington, D. C.***

COMPUTING AT LOS ALAMOS, GROUP T-1*

Max Goldstein

UNIVERSITY OF CALIFORNIA
Los Alamos Scientific Laboratory
Los Alamos, New Mexico

Computing has played an important role at the Los Alamos Scientific Laboratory. We run a somewhat informal computer installation whose chief function is the solution of scientific problems. Since we like to think of our equipment as another useful tool that aids, more or less, in this task, we are not enamored of routines for their own sake but as they help solve problems. I would like to tell you a little about our group, our method of operation, and about some of the routines we have found useful.

Group T-1 was once referred to as the 701 group, we were then promoted to the 1402 group and now, with our two 704's and one 701, I suppose we should be called the 2109 group. In September we should advance again when we receive our third 704.

The group consists of forty-six people at the moment with Bengt Carlson as group leader, myself as alternate group leader, and Ed Voorhees as head of the section on utility programs. About thirty persons whose educational backgrounds range from BA to PhD do at least some programming for the 701 and 704's. About eleven persons perform the routine functions connected with the operation of the installation such as the keypunching and verifying of cards, assigning of machine time, dispatching, night operating when requested. The IBM maintenance engineering staff of fifteen is not regarded as being part of the group.

Our first 701 went into operation in April 1953, our second one in December 1953, our first 704 in January of this year, and the second one in June. The machines are in operation three shifts daily except Sunday. On the 701's, we have logged a total of 32,276 hours and, on the 704, 5,375 hours as of the end of May. More than seven hundred individual problems have been completed including ones in neutron diffusion, weapons, power reactors, nuclear propulsion, controlled thermonuclear energy, nuclear physics, and mathematics.

Our group uses the so-called open-shop policy. By this method anyone can code and get on the machines. This method of operation, of course, requires that advice in numerical analysis and assistance and instruction in coding and machine operation be made available to anyone in the laboratory. It also means that a library of routines is needed so that the machines may be used conveniently and efficiently by people who are not "experts." We adopted this open-shop policy mainly because we were convinced that it would not be efficient, in a laboratory staffed by physical scientists, for our group to code all problems even if our personnel were adequate. To date more than two hundred and fifty people have done at least one problem on the machines. Of course not all problems are done by the originator, and it still is the major responsibility of the group to analyze, program, and code large problems. Selection is made on a priority basis according to the availability of personnel, and problems are accepted in varying stages of development.

The fact that we were, more or less, in the forefront in the use of this type of high-speed digital computer meant that we could not draw on anyone else's experience or use anyone else's routines. We are, of course, all in favor of avoiding redundant effort and we do belong to SHARE; however, when we began planning for our 701 and later for our 704, there were not many programs to be shared, so we have been more in the position of contributor than receiver so far.

*Work discussed in this paper was performed under the auspices of the AEC.

As far as automatic programming goes, we have given it some thought and in the scientific spirit we intend to try out FORTRAN when it is available. However, our future effort in programming research will be directed to our next generation machine, for which we are now negotiating, a machine about a hundred times faster than the 704. We will have an opportunity to influence the machine logic and instruction system, an area which, we feel, will greatly facilitate the development of efficient computer-assisted programming.

I would like to turn now to some of the routines our group has developed. Codes designed for particular physical or mathematical problems, however, are not included.

I am going to deal mainly with 704 routines, but I would like to mention one 701 routine which was developed at Los Alamos that proved extremely useful. This is the Dual coding System,¹ devised in 1953 primarily as an interpretive routine to provide a flexible and fast mechanism for problems requiring the use of floating-point operations. Its second purpose was to provide a floating-point system that could be used conveniently in conjunction with fixed-point calculations. The main features of this system are the great ease with which the programmer can alternate between the use of standard machine commands, and the strong similarity of structure between the floating-point orders and standard machine commands, since DUAL is entirely one-address in nature.

The 704 utility programs are classified into three categories: Subroutine (S), Debugging (D), and Auxiliary (A). The distinctions between the categories are as follows:

Subroutine - Programs in this category may be regarded as becoming fused with the programmer's code since such codes have coded references to the subroutine. Examples of subroutines are decimal-print programs, math functions, and a punch program intended to punch problem results. Two characteristics of such programs are that they are almost always entered by basic linkage (TSX) and they are assembled by the programmer.

Debugging - Programs in this category are used only to detect errors and will not do any correction of errors. Tracing programs and memory-print programs fall into this category.

Auxiliary - This category contains miscellaneous programs which assist in the operation of the problem, the correction of the problem, or the interpretation of the problem. Included in this category are such programs as binary and octal loaders, assembly programs, control card punch programs, and mathematical abstractions. Programs in this category will not, in general, need to be assembled and will use control cards (if needed) rather than basic linkage.

I will briefly mention the programs, to date, in each category. If more information is desired, I will be glad to send details upon request.

The method of coding we use for the 704 is controlled by our assembly program (A 870). We feel that this regional-symbolic method simplifies the coding procedure and reduces the clerical work in programming. In addition to minimizing coding errors, we have tried to make reassembly, making insertions and deletions, convenient. A square root program is outlined in Fig. 1.

If the experience at other installations coincides with our experience at Los Alamos, I believe we may agree that the main bottleneck in the course of a large problem is the period beginning after the coding of the problem has been assembled and ending with the successful calculation of the first correct results — the debugging period. Also, in some problems, the code will never take on a fixed form; these code modifications call for recoding and debugging. We have therefore attempted to inculcate on our "customers" an appreciation of correct debugging techniques and have made available programs which, we hope, will facilitate this phase of programming.

Our philosophy with mathematical subroutines has been to make them as general as possible in keeping with the concept of a mathematical formula; that is, it is completely general until specific values of variables are inserted, limits given, etc.

An index to our current 704 utility programs follows.

¹S. I. Schlesinger, "Dual Coding System," Los Alamos Report LA 1573, July 1953

704 Regional-Symbolic

1	2	3	4	5	6	7	8
SQ							
ROOT							

LA A 870

LA A 870

UNPUNCHED REMARKS	LOCATION				OPERATION				ADDRESS				DECREMENT				DATA / REMARKS			
	SEQUENCE F				F				F				F				F			
	18	19	20	21	18	19	20	21	22	23	24	25	22	23	24	25	26	27	28	29
ASSUME ENTRY BY: 0.005 0000					12				000	0001	0	0000	0000	0000	0000		SQUARE ROOT			
QTSX 5.0 C					1				2	32							ERROR STOP			
WITH ARGUMENT IN AC					1				1	0	0						STORE INDEX			
EXIT WITH 20.75 IN AC					2				1	1							STORE ARG			
					3				NR	2	30									
					5				LR	0	1									
					6				A	1	1									
					1				LR	0	1									
					8				A	2	31						64 1/2 AT 8			
WITH THIS INITIAL					9				SAX	2	32	0					3 TO INDEX C			
GUESS WE NEED					12				ST	1	2									
ONLY 3 ITERATIONS					13				CA	1	1									
FOR CONVERGENCE					14				ED	1	2									
					15				SQ	1	3									
					16				CA	1	3									
					17				FA	1	2									
					19				S	2	30						DIVIDE BY 2			
					18				TX	2	12	0	0	1						
					19				SAX	1	0	0					RELOAD C			
					20				T	0	1	0					EXIT			
					30												0.0100000000			
					31												1.0040000000			
					32				H	0	3									

Fig. 1 - Illustrating the regional-symbolic method for a square root routine

AUXILIARY ROUTINES

A 020	Self-Loading Binary Loader
A 021	Binary Loader for 701 Data
A 022	Upper Binary Loader
A 027	Load Binary Correction Cards
A 028	Insert Break Points
A 029	Data Spreader
A 080	Octal-Alphabetic Instruction Input (1 per card)
A 081	Octal-Alphabetic Instruction Input (4 per card)
A 082	Octal Data Input (4 per card)
A 220	Binary Punch
A 221	Binary Punch which Omits V, R, and Check Sum
A 321	Reproduce Tape in Binary
A 420	Dump Memory on a Selected Logical Drum
A 620	Binary Drum Corrector
A 720	Reproduce Binary Cards with Correct Check Sum
A 721	Check Binary Cards without Destroying Memory
A 722	Check S-Deck
A 870	Assembly Program
A 873	Print Origin Table
A 900	"Package" Floating-Point Polynomial Solver

DEBUGGING ROUTINES

D 070	Selective Tracing Program
D 080	Logic Trace
D 081	Logic Trace with Partial Print
D 480	Trap Memory Print
D 481	Dynamic Print Monitor
D 620	Transfer Search Program
D 621	Search Memory
D 622	Compare Memory with Binary Cards
D 770	Memory Print
D 771	Floating-Decimal Memory Print (7 per line)
D 772	Floating-Decimal Memory Print (10 per line)

SUBROUTINES

S 000	Fixed-Decimal Data Input (4 per card)
S 001	Fixed-Decimal Data Input (7 per card)
S 010	Floating-Decimal Data Input (4 per card)
S 011	Floating-Decimal Data Input (7 per card)
S 020	Basic Linkage Binary Loader
S 021	Binary Loader for 701 Data
S 070	Fixed- and Floating-Decimal Data Input (7 per card)
S 100	Fixed-Decimal Print (6 per line)
S 101	Fixed-Decimal Print (10 per line)
S 110	Floating-Decimal Print (17 per line)
S 111	Floating-Decimal Print (10 per line)
S 113	Floating-Decimal Print (1 line, 7 per line)
S 114	Floating-Decimal Print (1 line, 10 per line)
S 220	Basic Linkage Punch
S 270	Identification Punch
S 321	Read and Write Tapes
S 420	Drum Write Read Routine
S 740	Dual-to-704 Floating-Binary Conversion

SUBROUTINES (Cont'd)

S 800	Floating-Point Square Root 24, 1.67, 8th
S 805	Floating-Point Cube Root 34, 4.6, 8th
S 810	Floating-Point Nth Root 68, 2.17 + 0.936N, 8th
S 815	Floating-Point Exponential 48, 3.48, 8th
S 816	Floating-Point Exponential 63, 2.616, 8th
S 820	Floating-Point Natural Logarithm 39, 2.22, 8th
S 821	Floating-Point Natural Logarithm 30, 1.33, 3rd
S 825	Floating-Point Sine or Cosine 54, 3.30, 8th
S 835	Floating-Point Arcsine 55, 4.848, 8th
S 840	Floating-Point Arctangent 36, 3.3, 8th
S 841	Floating-Point Sinh and Cosh 54, 3.732, 8th
S 850	Square Root of a Complex Number 70, 3 4, 8th
S 851	Complex Multiplication 16, 1.2, -
S 852	Complex Division 39, 2.1, -
S 853	Natural Logarithm of a Complex Number 117, 6.9, -
S 854	Exponential of a Complex Number 125, 10.9, -
S 855	Sine or Cosine of a Complex Number 139, 11.2, -
S 856	Sinh or Cosh of a Complex Number 145, 11.3, -
S 860	Floating-Point Natural Logarithm of the Gamma Function for Floating Complex Arguments 272, 7, 6th
S 871	Floating-Point Polynomial Solver 720+, -, -
S 872	General Least Squares 240+, -, -
S 880	Floating-Point Linear Interpolation 28, -, -
S 885	Matrix Equation Routine 103, 30000, -
S 886	Floating-Point Modified Runge Kutta 85, -, -
S 887	Integration of Second-Order Equation with First Derivative Absent 80, -, -
S 896	Clebsch-Gordan Coefficients 393, 18, -
S 900	Fixed-Point Square Root 38, 2.36, -

The numbers following the math routines indicate the following:

Length (not including region 1)	Maximum time in ms (assuming 7 cycles for FA and FS)	Error (first nonsignificant digit)
------------------------------------	--	---------------------------------------

CODING FOR THE MANIAC*

Mark Wells

UNIVERSITY OF CALIFORNIA
Los Alamos Scientific Laboratory
Los Alamos, New Mexico

In addition to the IBM 701 and 704's activities already described by Max Goldstein, Los Alamos has built and operated the Maniac series of computers. I would like to discuss and evaluate three coding schemes which we have developed for this series. First to be described will be a relative or descriptive coding scheme which has been in use with Maniac I for three years, second will be a similar but improved scheme prepared for use with Maniac II, and third will be a formula coding scheme also to be used with Maniac II.

Before launching into the coding discussion, we should describe very briefly the computer itself, as it is obviously important to understand the idiosyncrasies of the monster for which the coding is to be done.

Maniac I is a high-speed, general-purpose, digital computer with electrostatic storage of 1024 forty-bit words and magnetic drum storage of 10,000 such words. It is a single-address, parallel computer. Number representation is fixed-point binary; input is with five-hole paper tape, through a photoelectric reader; output is to paper tape, teletype printer, and a fast line printer. Manually positioned magnetic tape is available for auxiliary memory.

A forty-bit word may be an operand, a pair of instructions, or both, according to the use made of it. An operand is usually a number with a sign bit and thirty-nine bits for magnitude. An instructional word consists of two half-word instructions, each having five tetrads of four bits each. The first two tetrads of an instruction constitute an order according to the vocabulary of the computer. The last three tetrads furnish an address (or some other number relevant to the particular order).

The vocabulary of Maniac I is reasonably standard, containing the basic add, subtract, multiply, divide, store, shift, substitute address, unconditional transfer of control, conditional transfer on non-negative, and the required input-output orders.

The natural grouping of bits of a forty-bit word is into ten hexadecimal characters or tetrads. Thus, the sixteen symbols 0 through 9 and A, B, C, D, E, and F are considered basic with regard to input and output. The line printer can print these sixteen symbols and a minus sign and a decimal point.

Order times (which include fetch of instruction and operand) are $80\mu s$ for add, 1 ms for multiply and divide, 45 seconds per 1000 words for paper tape input, 1.6 seconds per 1000 words for drum-to-electrostatic memory, and 600 characters per second for the line printer.

I would now like to begin the discussion of the descriptive coding scheme for Maniac I. Similar coding on other computers is called relative, or regional, or symbolic coding. Early in 1953 it became clear to the Maniac group (as it has to many others in the computer field) that much of the bookkeeping detail of writing a code could be handled more efficiently by the computer itself. Our primary objective was to eliminate so-called blunder errors: miscopies,

*Work discussed in this paper was performed under the auspices of the AEC.

miscounts, and mistransfers, particularly when changes were made in a problem code. The fundamental premise was that linear, independent sections of code, in our case boxes of a von Neumann type flow diagram, are easy to code without error. Furthermore, because of the independence, any relatively coded box could be altered without affecting the rest of the code.

At that time an assembly routine was written, but acceptance was only halfhearted until, with the acquisition of our 10,000-word drum in late 1953, an elaborate, far less restrictive assembly routine was written. The employment of descriptive coding in our group has been near 100 percent since that time.

The word, descriptive, derives from the method of labeling the various storage types. Thus, storage for addresses is labelled with an A, binary constants with a B, constants in decimal with a C, dynamic or temporary storage with a D, and flagged instruction, i. e., variable transfers, with an F.

It is considered desirable to have the format of a relative or descriptive code similar to that of an absolute code. Therefore, five-tetrad descriptive instructions are used. The first two tetrads constitute an order from the standard Maniac I vocabulary. For instructions referring to storage the third tetrad is the storage type label and the fourth and fifth tetrads give the hexadecimal position in that storage block of the operand. For instructions referring to groups of instructions, called boxes, the third and fourth tetrads give the box number, and the fifth tetrad the instruction number of that box, usually 1. Absolute addresses may be included in most instructions since the largest Maniac I address in hexadecimal is 3FF and such relatively low numbers are left undisturbed by the assembly routine. Subroutines from the Maniac's extensive library, which are included in a certain problem, are merely given box numbers for that problem and are then transferred to as such. Their tapes are included directly at the end of the descriptive tape. Subroutine storage is amalgamated with that of the main problem in the sense that constants are compared and duplications omitted, while subroutine dynamic storage uses the temporary storage locations of the main problem.

The descriptive tape, which is the input of the assembly routine, consists of the five-tetrad instructions of the descriptive code grouped into boxes, preceded by the corresponding box numbers; the input storage grouped according to type; information words telling where the absolute code should be constructed; and, finally, the subroutines. The assembly routine pairs up and locates the instructions, translates addresses to absolute, and converts the input numbers. Output is a printed copy of the absolute and descriptive code and a magnetic-tape record or, optionally, a paper-tape punch of the absolute code. This computer ready code is also put into the electrostatic storage in case debugging is to begin immediately.

Descriptive tapes are only trivially longer than corresponding absolute tapes. Assembly time, including read, print, and record, runs from less than a minute for short service routines to about five minutes for a 1000-word code, with, say, 1000 instructions and 500 numbers.

The assembly routine itself is 1500 words long; hence, it works with the auxiliary drum storage during assembly of a problem. Approximately six man-months were required to develop the scheme to maturity, a state it has held for two and one-half years. It is extremely difficult to estimate its value in terms of man and machine hours saved. Let's say problem preparation time is reduced by 40 percent and debugging time by about 30 percent.

I would now like to introduce a new computer, Maniac II, for which improved coding schemes have been and are being devised. Maniac II is again a high-speed, general-purpose, single-address, parallel, binary, digital computer, but now we have electrostatic storage of 12,000 forty-eight-bit words. It operates in floating- or fixed-point and has automatic address modification by means of three index registers, sometimes called B registers. Input is via paper tape through a photoelectric reader, computer-controlled magnetic tape, and a typewriter. Output is to magnetic tape, punched paper tape, the typewriter, and a fast line printer.

The forty-eight-bit word, when considered a number, has a sign and three bits for exponent of the floating-point base which is 2^{16} , and sign and forty-three bits for the fraction. The number range is thus 2^{-155} to 2^{112} . Numbers with exponent zero are equivalent to fixed-point numbers, a fact which simplifies computer hardware. In this case, the half-word instruction

contains six tetrads, two tetrads for the order, two bits to select the B register (where the zeroth B register is a mythical register defined as containing zero at all times), and fourteen bits for partial address or other number relevant to the particular order. The partial address is added to the contents of the selected B register to furnish the complete address.

Vocabulary extensions to that for Maniac I include the necessary floating-point orders, square root, extract, set, count, and compare for the B registers, conditional transfers on zero and overflow, a pathfinder transfer for use in returning from subroutines, sense and set sense, and the magnetic tape input-output orders. Primary input is by seven-hole paper tape through a photoelectric reader. As in Maniac I, external control of the reader is by a load memory switch. Internal control is by the orders "read word" and "read hexad," that is, read one paper tape character. This latter order is extremely useful during automatic coding, when the computer is constructing a code from an arbitrarily grouped set of tape characters. The line printer can print the usual sixteen symbols: 0 through 9, A through F; and a minus sign; decimal point; and asterisk. The punch can be ordered to punch a word consisting of twelve tetrads or one six-bit character, a hexad.

The order times are about $20\ \mu\text{s}$ for add (including most floating adds) and $200\ \mu\text{s}$ for multiply. The magnetic tape transfers 250 words per second; the reader is twice as fast as before, and the printer again is capable of 600 characters per second.

The philosophy and basic procedure of the descriptive coding scheme for Maniac II is the same as that for Maniac I. Naturally enough, improvements have been made. Besides generally increased flexibility, there are three main changes: Firstly, in order to separate B register reference from the address tetrads, seven-tetrad descriptive instructions are used. This also permits the use of more distinct storage blocks, and each may contain more quantities. Secondly, the flagging, that is, marking for reference, of specific instructions, which before was limited to variable transfer orders and done within the descriptive instruction itself, is now accomplished by means of a four-tetrad flag preceding the instruction. This is useful since any instruction being referred to by another instruction may be flagged, thus obviating a detailed count of instructions within a box. Furthermore, by choosing thoughtfully the numbers to be used as flags, a programmer may use them to organize his code into natural subdivisions. Thirdly, since the magnetic tape for Maniac II will be equipped with searching facilities, the commonly used subroutines can be stored on magnetic tape as part of the assembly routine. Thus, to incorporate a subroutine in a problem, the programmer need merely refer to its call number. Less common subroutines may be incorporated as before.

Perhaps four man-months have been used in the development of the assembly routine. Another two man-months will be needed for debugging and polishing when Maniac II goes "on the air" toward the end of the summer. The routine is 2000 words long. Problem assembly time will certainly be less than before because the computer is faster and has a much larger electrostatic memory. In this case, descriptive tapes will run perhaps 30 percent longer than corresponding absolute tapes.

Our final discussion concerns a coding scheme developed for Maniac II, which falls in the realm of automatic of formula coding. The primary motivation, of course, is to shorten the time spent in problem preparation, including debugging. The approach is to let the more reliable computer do the translation, via an assembly routine, from mathematical formulation to computer code. This computer code is to be timewise efficient to accommodate production-type problems and spacewise efficient for compatibility with experimental problems which are continually changing.

The mathematical formulation of a problem consists of an ensemble of defining equations, control equations, and information equations. A flow diagram is one possible example of such an aggregate. A seven-hole paper tape containing the sequence of equations serves as input for the assembly routine. Output is a printed-copy and magnetic-tape record of the absolute computer code, along with a punched paper tape listing storage assignment, and pertinent comments to be printed when one is off the computer.

Unfortunately, I cannot discuss in detail the ground rules for formulating a problem nor the notations we have adopted. It should be remarked however, that our philosophy has been that self-explanatory notation is superior to concise notation whenever there is a disparity

between the two. The scientist should be able to understand, without special study, the equations serving as input.

Only after much more experience is obtained can one adequately evaluate schemes such as this. There should be great savings with regard to personnel training, problem programming, and problem debugging.

Tapes using this formula notation will be less than one-half the length of corresponding tapes for problems coded in absolute computer code. A rough estimate of assembly time is 8 minutes for a code containing 1000 instructions and 500 numbers. The assembly routine will not exceed 4000 words of code. It contains as subroutine the relative coding assembly routine previously discussed.

The progress made in the last five years on the design of digital computers, as we all know, has been astronomical. Unfortunately, problem preparation has not kept pace. To close the gap, our group now has the equivalent of two full-time people working on coding techniques for the new Maniac. However, with the advent of formula-type coding, distinct computers begin to lose their individuality, at least from the programmer's point of view, making unified intergroup activity desirable. The day when a problem may be formulated in a universally acceptable notation, and then coded and solved by any computer is to be anticipated. Our group will welcome intergroup communication on these matter.

PROPOSED ADVANCED CODING SYSTEM FOR UNIVAC-LARC

Frances E. Holberton

*David Taylor Model Basin
Washington, D. C.*

A DESCRIPTION OF THE UNIVAC-LARC

The UNIVAC-LARC is a large-scale, high-speed, internally stored program, digital computer being developed by the Remington Rand UNIVAC Division of Sperry Rand Corporation. The primary purpose of the computing system is to solve large problems in the field of science and engineering. Because the problems which face the Bureau of Ships encompass the fields of business and logistics as well as science and engineering, it is envisioned that the UNIVAC-LARC will be used as a completely general purpose high-speed computer.

The basic computer system contains two computers. One computing system is designed to perform the arithmetic and logical functions of the system on information stored in the main core memory of the computer. The computing unit is a parallel computer capable of performing both fixed- and floating-point arithmetic operations in a binary-coded decimal form.

The second computing system is an input-output processor, designed primarily to provide flexible, parallel, and coordinated control of the input-output equipment. Except for intercommunication facilities, the computing unit and the input-output processor operate independently.

The high-speed magnetic core memory is shared by the processor and the computing element. The memory is divided into units, each capable of storing 2500 computer words of eleven decimal digits and a sign. Up to thirty-nine units or 97,500 words may be incorporated into the system. The Bureau of Ships contemplates a magnetic-core memory of 20,000 words.

The high-speed memory is backed up by a magnetic drum file memory and a tape servo system. Up to twenty-four magnetic drums, each capable of storing words of twelve decimal digits, and a maximum of forty tape units may be incorporated into the system.

The main computing unit is controlled by a program stored in the magnetic-core memory containing a single instruction, or command, per word. Each instruction word of twelve decimal digits is made up of an operation code, which specifies the operation to be performed; an M address, which specifies the memory location of the operand involved in the operation; a B register address, which specifies one of a number of index registers which modifies the M address of the instruction as the instruction is read out of the memory; and an A register address, which specifies one of a number of arithmetic registers used as accumulators for storing operands and operation results. The instruction word format allows three digits for the operation code, two digits for the A register address, two digits for the B register address, and five digits for the main memory address. Up to a maximum of a hundred A registers and ninety-nine B registers are possible within the framework of the instruction format.

The use of multiple A registers provides what amounts to a two-address instruction on many operations. This feature, together with the addressable B registers, gives the programmer the flexibility and speed required to generate optimal programs.

Various errors due to improper scaling of numbers and exponents, nondecoded operation codes, or computer malfunctioning will produce a control jump to specified memory addresses, for which programmed subroutines must be provided to either halt the computer or rectify the error.

The processor computer, although primarily designed to handle information transferred from drums, tapes, printers, and any other directly connected input-output devices to the main core storage, can in some respects be considered a more general purpose computer than the main computing element. It is possible to use the processor, by means of a stored program, as an interpretative system for the operation performed by the main computer. The reverse is not possible, since the main computer cannot directly call for information from drums, tapes, or any auxiliary input-output unit without the use of a subroutine stored and executed by the processor.

In normal operations, however, the processor can be considered to be a slave to the main computer. The processor, unless instructed to perform some function, will essentially mark time waiting for a signal from the main computer.

The information transferred from the main computer to the processor by way of the transmission system can be considered in the form of a pseudocode, since the information does not enter the control element of the processor directly. The information transmitted is decoded by a programmed subroutine to channel the operation and variables to the correct stored subroutine for processing. When the necessary processor instructions are initiated to set the desired functions in motion, an additional parallelism is set into play. By the use of synchronizers, which are scheduling, timing, counting, and decoding devices associated with each different type of auxiliary equipment, the processor is able to channel specific instructions to a selected synchronizer to perform the basic information-transfer operation.

Once the processor program has set a synchronizer into play, the processor is available to perform a new operation or to monitor the ones already in process. The processor is able to receive a sequence of signals from the main computer and effectively stack the operations in a list. When the list is full, the main computer is interlocked against transmitting to the processor until one operation has been completed and erased from the list. Although the processor is able to initiate only one input or output function at any one time, once the appropriate synchronizer has received its information another synchronizer can be set in operation for the next instruction.

The amount of parallelism performed in the processor is dependent upon the number of synchronizers purchased and efficient sectionalizing of the memory. It is possible to read information from two different drums to the memory, write onto a third drum, read or write on as many as four servos, and print on the console printer, all at the same time. Whenever the processor is making use of a section of the core memory, the main computer is interlocked against using the same section until the data transfer has been completed.

Since the main computer and the processor both make use of the same core storage and transfer lines, the basic timing cycle is apportioned on a time slot basis, so that each computer operates at maximum speed.

Although two computers and a number of synchronizers may be in operation simultaneously, the concept need not complicate the mathematician's programming. If the processor is considered to be, in logic, nothing more than a library file of special-purpose subroutines which the programmer calls into operation by the main routine, the programmer can forget the presence of a second computer. The programmer can consider the transmission instructions as built-in hardware if the concept of parallel-operated subroutines is a mental stumbling block.

It is expected that the programmer will make use of a processor subroutine library which will be set up by a special programming group. The order structure of the processor is, by necessity, different from the main computer, since special instructions are required for the input-output equipment. It is not necessary for the processor to contain floating-point operations, multiplication or division instructions. The processor can be programmed to perform these operations if it is to be used to diagnose errors in the main computer. Since all registers

including the control and computing registers are addressable, the processor can be used as an efficient diagnostician.

PROGRAMMING DEVELOPMENTS AT THE DAVID TAYLOR MODEL BASIN

The Applied Mathematics Laboratory at the David Taylor Model Basin operates as a service bureau for the Bureau of Ships. At the present time two UNIVAC I's are in operation at the site. The programming staff, however, prepares problems for the UNIVAC I, UNIVAC II, NORC, and the IBM 704. A staff which is versatile enough to program for four computers will find the transition to UNIVAC-LARC a relatively simple one.

The range of automatic coding systems can run the gamut from the completely artificial system which camouflages the computer and programming techniques by means of a very versatile pseudocode to a system which makes use of only machine code with a library coded in machine language and relative addresses.

The versatile pseudocode system can be used by both the novice and the experienced programmer, but its success with the experienced programmer is dependent upon the foresightedness of the group developing the pseudocode. When a computer has been in use for many years, it is easier to make an evaluation of the best programming and coding techniques and incorporate them into the system, than to attempt the same project on a new computer. I doubt if a single pseudocoding system developed during the early design stages of the computing equipment is still in daily use. The systems which are in use today are the result of actual experience on the equipment. No matter how much the systems' developer tries to project himself into the future, he is bound to be shortsighted in some area, since his experience has been gained on earlier equipment.

The programming staff at the David Taylor Model Basin, at present, operate in a closed shop, but over the next few years this may change, due to the difficulty in hiring programmers and the increased speed of problem solution. In order to satisfy the appetite of UNIVAC-LARC and keep it constantly fed, it may be necessary to have a very versatile pseudocode which will permit the programmer to code in equation form, so that the one-time programmer need not be confronted with the complete flexibility which is built into the LARC design.

If such a system is developed, it should be in the form of a translator, whose computer output is the input to the earlier developed compiler system. In this way the library of subroutines developed over a period of time could be used without reprogramming a new one.

I wish to turn now to the immediate problem of developing an automatic coding system which should be available to the programming staff well in advance of the computer delivery date.

Since the success of any automatic coding system is dependent upon the acceptance of the chosen system by the potential users, their desires and wishes should be taken into full account. The experienced programmer who is used to producing hand-tailored coding is skeptical about the whole idea of automatic coding and poses four fundamental questions which he would like to have answered:

1. "What advantage is the system going to offer me over a hand-tailored code? I consider myself as good a programmer if not better than the fellow who is designing the system."

This is a very valid question since, if the advantages of the system do not outweigh the disadvantages by a large proportion, the system is certainly not a good one.

2. "Is the system going to restrict my programming techniques? I pride myself with being able to code short, concise routines which require no waste motion on the part of the computer. Is this system going to inhibit any initiative on my part to develop better techniques?"

No completely artificial pseudocode can foresee all the combinations of computer instruction arrangements that a programmer will dream up to his advantage. The designers of the equipment themselves would be surprised at the uses programmers have found for various functions. In order to supply a negative answer to this question, it is necessary to include the computer code as either the basis of the pseudocode or as an adjunct.

3. "Will the system make it harder and not easier for me to check out a problem on the computer? Much has been said and written about automatic coding eliminating 90 percent of the code-checking processing, but I already eliminate that 90 percent by doing a thorough desk check of my coding. I am interested in that 10 percent which is sometimes even difficult to find in hand-tailored coding."

If the system is set up so that there is a close resemblance between what the programmer codes and what the compiler produces, the code-checking process should certainly not be more difficult. The compiler can be made to produce an edited copy of the coding with columns for the original pseudocode and corresponding columns for the compiled program.

4. "Who is going to set up and maintain the library of subroutines?"

This is certainly an appropriate question since, in an installation where the pressure of work to be done is always greater than the supply of people to perform the job, a lower priority task such as creating subroutines may get lost in the shuffle. If the subroutines which are to be filed in the library for future use are programmed in the same pseudocode language as the main routine, no special rules must be learned to set up subroutines for the file. The problem is then reduced to one of incorporating the new subroutines into the permanent file.

In order to ease the mind of the experienced programmer, I would like to put forth a list of specifications for a "programmer's" automatic coding system and have him tear it apart, making deletions or additions.

The system must present some advantage as a coding system, without relying completely on the library for its entire asset. Since problems are expected to become larger and more complicated, many programmers will be handling different parts of the same problem. The consolidation of their effort into one unit must be possible with a minimum of preplanning. In order to accomplish this, a flexible relative addressing system is required. Although the computer is basically a decimal computer, the processor is capable of converting the seven-pulse alphanumeric code created by a Card-to-Tape Converter or a Unityper into a two-decimal digit code, and performing the reverse upon command to transfer alphanumeric data for printing on auxiliary equipment. Since five digits are provided in the address portion of each instruction to specify the absolute memory address, it is possible to use a relative addressing system which incorporates alphabetic characters to regionalize the sections of the coding. In this way a large problem can be broken up into many hundreds or even thousands of small parts, without duplicating the relative addresses.

I feel that the programmer should not be restricted to using the alphabet in a sequence. If he wished to omit certain alphabetic letters or use a double or even triple lettering system for his relative addresses, the compiler should not balk at this. It should be sufficient to label each section or part of the coding with the starting address composed in an alphanumeric word and supply an ending word to each section. The label word and, if specified, the ending word will be omitted during the compilation. Constants may be stored in three different ways. If they are stored after the end-of-section word, they are given a relative address. Constants which can be expressed in a few digits may be written directly in the instruction word in place of the relative address. Constants which require more than a few digits may be stored directly below the instruction using a special symbolic address within the instruction word.

There should be no restrictions on cross referencing any routine, even though filed in the library. It is necessary only to precede the instruction making use of a cross reference with the name of the subroutine. The programmer may cross reference between memory loads or overlayed parts of a problem. He may, if he so desires, specify the starting absolute address of any subroutine of his own problem.

The operation part of the pseudocode shall include the complete machine code for the main computer as its basic element. Whether the decimal digit operation code should be expressed in an alphabetic pneumatic code is debatable and should be left to a majority vote. The choice makes little difference to the compiling method. Additional instructions in an alphabetic form will be required to express return jumps to closed subroutines, linkage to parameters, variables, and any pseudocodes which may be deemed necessary to preserve parts of instruction words normally destroyed by address modification.

Each problem which is coded can be considered to be a subroutine, and as such may be filed in the library without revision. Any subroutine which is filed in the library may have as few as two elements or as many as five.

The first element present in all subroutines is the problem description. The problem description may contain only the name of the subroutine or it could contain a complete description of the problem method, name of the programmer, date programmed, in a format for printing on auxiliary equipment. The other pertinent element is the coding. When a routine contains only these two elements, it is considered to be a static subroutine containing no parameters, and will be compiled only once regardless of the number of references made to it.

Subroutines which are filed in the library may call on other subroutines without the knowledge of the user of the higher level subroutine. It is necessary for the user to know only the name, the function performed by the high-level subroutine, and any parameters required when making use of the named routine. When a subroutine requires parameters or variables to be supplied by the user, it becomes a dynamic subroutine. A dynamic subroutine may make use of another dynamic subroutine from the library. Each subroutine which uses a lower level dynamic subroutine has two options available at the time the higher level routine is programmed. The higher level subroutine can supply the necessary parameters at this level, which to all practical purposes produces a static subroutine, or it can defer the specifications to the next higher level subroutine.

If, for instance, a high-level subroutine **A** calls on subroutine **B**, **B** calls on **C**, and **C** calls on **D**, and each subroutine has two variables, the user of subroutine **A** is requested to supply eight variables.

When subroutines contain variables which must be supplied, an additional element called linkage is incorporated into the subroutine which uses the lower level subroutine requiring the variables. For instance, when subroutine **C** was coded, a linkage section was supplied naming routine **D** and specifying a deferment of these two variables. The descriptive part of routine **C** will state that four variables are required. When routine **B** was coded using routine **C**, a linkage part was added to **B** specifying the name of **C** and a deferment of four variables. It is not necessary for routine **B** to know that two of these variables are used in **C** and two in **D**, or even that **D** is present at all. By the use of linkage, variables may be supplied or deferred to the next higher level. There is practically no limit to the kind of deferment which can be made. It is possible to defer even the name of subroutine, or to defer the selection of parts of routines. In this way a problem can be changed each time it is compiled by naming the variable parts or the choice of method or subroutine used.

When the programmer makes use of a dynamic subroutine within his own problem and does not wish to file this problem in the library, he will probably supply all the variables in the linkage element within his routine. If, on the other hand, he would like to file the problem in the library for future use and still maintain the flexibility built into his system, he may defer all or part of these variables up one more level. Since the problem is to be compiled, the variables must be supplied somewhere. When variables are deferred in the subroutine written for compilation, it is necessary to precede the subroutine with the values for all deferred variables. This part is called the zero linkage or highest level linkage.

When the subroutine is placed in the main library file, the zero linkage is automatically omitted and the subroutine becomes a dynamic subroutine again without any modification made to the basic subroutine.

In order to allow generative-type subroutines to be placed in the library, an additional element called a preset part is required. The preset part is basically a generator which makes use of specifications stored in the linkage element to create a new and perhaps extensive linkage element, or to set up to select parts of coding for later compilation. The preset part can be set up to unwind routines. It is basically used when the length of a routine, other than selection of specific parts, is not predetermined.

Some subroutines may require a fifth element called a postset part. The postset operation is performed on the compiled coding to fabricate constants which make use of absolute address or any operation which can be performed after compilation.

The instructions used in the preset and postset operations are not compiled as part of the running problems. Subroutines may be called from the library either as open or closed routines. When a problem makes use of a subroutine once in only one sequence it should be possible to compile the subroutine directly into the sequence, omitting the entry and exit jump control elements.

Any references within the main computer subroutines to operations performed by the processor will cause the compiler to supply the appropriate processor subroutine information to the compiled problem. It is envisioned that all frequently used processor subroutines will be stored permanently on a drum. When a problem is compiled, only the name and the necessary parameters for a processor subroutine will appear at the beginning of the compiled routine. When the problem tape is run on the computer, the first operation performed will cause the processor to select the necessary subroutines from the information stored on the tape and assemble the routines from the drum. In this way, the number of instructions stored on tape can be kept to a minimum. A compiled processor routine may use a full memory unit of 2,500 words or more if many types of auxiliary units are used, so that it becomes desirable to assemble this information at the time of running, rather than read it from tape. By exercising a special option, the programmer may have the complete processor subroutines written on the tape with the compiled program.

In order to allow complete flexibility to the coding system, it will be necessary to supply a zero-level linkage element to most problems to be compiled. It is necessary to supply all the subroutines required for the error routines in the final compiled program. Since no direct reference is made to these subroutines within the problem, it is necessary to supply a list of desired error subroutines in a linkage element. The programmer should be able to specify the error subroutines desired as a package or by separate subroutine names. If the subroutines are specified as a package, the programmer should be able to call for a certain package group, but specify the name or names of subroutines to be omitted from the package and any special ones inserted. Undoubtedly, after the computer is put into use, certain error subroutines will become standard, and it should not be necessary to specify each individual error subroutine since many are required. The package elements will be stored as specification lists on the drum, much as a subroutine is stored, and new package groups made up as the need develops. If no zero-level linkage is supplied with the program to be compiled, the compiler will select a fixed package set of subroutines for the processor.

The problem of designing a compiler system for the UNIVAC-LARC is not materially complicated by the presence of two computers. Difficulties, however, do arise from the existence of many arithmetic and index registers. In order to allow complete flexibility to the use of these registers, it will be necessary for the programmer to specify the arithmetic and index registers as containing expendable or nonexpendable information. When a subroutine jumps control to another subroutine in the library and nonexpendable information is stored in some of the A or B registers, the programmer must supply the necessary coding to store this information before entering the subroutine and to restore the contents to the A and B registers after a return jump. A pseudocode word could be used to store as many as five registers within a single twelve-digit instruction word to reduce the number of instructions required to perform this function. During the compilation, the pseudo word would be converted to LARC "store" instructions. This additional coding will be omitted from the compiled problem if the lower level subroutines which are called upon do not make use of the total number of registers provided. The nonexpendable register allocation may be specified in the zero-level linkage as a priority list, so that the computer can make a reasonable assignment. If a priority list is not supplied, the compiler will assign registers in the order of occurrence. By assigning non-expendable and expendable storage register relative addresses, the programmer need not concern himself with registers used by the lower level subroutines stored in the library. When a subroutine stores information in an arithmetic or index register for use by a lower level routine, and the relative addresses used for these registers are not identical, the programmer must supply the name of the lower level routine and the relative address equivalent in the linkage element of the higher level routine, so that the compiler will make the correct absolute address assignment. Because the arithmetic and index registers may be addressed either by a two-digit address or by a five-digit memory address, it will probably be necessary to reserve a certain type of five-digit relative address structure to be used when referring to these registers in the memory address portion of the instruction word.

The Bureau of Ships is not purchasing the maximum amount of memory, drums, servos, and registers as an initial system. However, we do not wish to close the door on the purchase of any additional equipment as the need arises; therefore, the compiler must contain a list of the number of each kind of equipment available for use. This list must be able to be changed without involving extensive modifications to the compiler system when additional equipment is purchased. The automatic coding system which most closely resembles this set of specifications is one prepared by Anatol Holt and William Turanski of the UNIVAC Applications Research Center at Sperry Rand for UNIVAC I, called Generalized Programming.

RCA APPROACH TO AUTOMATIC PROGRAMMING FOR COMMERCIAL PROBLEMS

John H. Waite, Jr.

*Radio Corporation of America
Camden, New Jersey*

The decision to support an automatic programming project at RCA was made in 1953 at a time when considerable information on automatic programming techniques was available. It was realized that subroutine libraries and compilers as they existed in the A-2 Compiler days were not satisfactory for commercial applications, which have many more constraints than scientific processes, and, therefore, require more involved narrative descriptions and more and shorter subroutines. There existed some doubt as to whether the library approach could be used at all, considering the low frequency of use of some of the subroutines and the resultant high program access. The technique of generation, however, appeared to offer a solution, especially if it could be used to simulate the processes of a human programmer in converting a well-defined problem into an efficient program.

Consider for a moment what a programmer does with a detailed problem flow chart in producing a program:

1. He blocks out the data formats for input to the computer and output to the printer.
2. He assigns high-speed memory areas for the program, input working storage, and output working storage.
3. He orders the input sequences so that the proper data pieces are available in the high-speed memory in an optimum manner for the desired outputs.
4. He writes the instructions to direct the computer to carry out the problem logic, which involves recomposition of information in memory working storage, with associated counter, and register logic.

If the computer is to simulate these steps, can any procedural generalizations be made? It is felt that the following can be stated without dispute:

1. The data formats must be defined and ordered.
2. The problem flow chart must be converted to a symbolic equivalent for computer interpretation.
3. A technique for memory layout of the data must be developed and integrated with a method for inserting the data addresses in the instructions.
4. A technique for memory layout of the program must be developed and integrated with a method for inserting the program control transfers.
5. Techniques for inserting all of the necessary machine controls and presetting the appropriate registers and counters must be developed.

6. Methods of reducing program access, optimizing problem logic, and, in general, minimizing time or storage requirements must be worked out.

Correlation of the above with actual programs implies that all addresses are either data-referring addresses, program control addresses, or program constants; and that all operations fall into one of the two categories: operations derived directly from the problem flow chart, or operations derived from particular machine logic requirements.

These implications lead to the hypothesis that, if the "machine logic" requirements are analyzed and organized in such a way as to provide a dictionary of rules of machine syntax, the symbolic equivalent of the problem logic can be extended by the computer to provide a program in the same way a programmer writes the program.

From these considerations, three key areas of study become apparent:

Defining the problem criteria - This includes the development of a narrative code language which provides a basis for easily converting the problem flow chart to a symbolic equivalent, the manner of specifying the necessary data format characteristics, and the method of defining the order of the data flow.

Defining the machine syntax - this included an analysis of the translation of the problem logic steps to the machine program, the separation of machine function from problem function, the associativity among computer operations in basic instruction chains, and the derivation of a dictionary of functional machine requirements.

Defining methods for program composition - This included conversion of the problem logic to basic operation code sequences, the extension of these code sequences to include the machine function requirements, the insertion of data addresses, the insertion of program control addresses, and the necessary program optimizing processes.

DEFINING THE PROBLEM CRITERIA

Returning to the defining of the problem criteria, the question may be asked, "What does a programmer need to know about a problem?" In a simple mathematical example such as computing the sigmas for the formula

$$\sqrt{\frac{\sum x_i^2}{N} - \left(\frac{\sum x_i}{N}\right)^2}$$

it is necessary to know:

the values for N and X_i (where N determines the number of X 's),
the form of N and X_i on input,
the form of 0 on output.

A general subroutine with floating addresses suffices to handle all desired values of X and N anywhere in the memory, if the most general form for X and N is chosen as a basis for the subroutine.

However, in a simple commercial example such as stock accounting, the multiple tape inputs contain many items requiring different operations, so that a subroutine written to handle one application would rarely satisfy another application, at least until some standardization of data formats and input selections is imposed upon the commercial field of users.

This does not mean, however, that automatic programming is not feasible in commercial problems. It means rather that the approach has got to be based upon a more discrete definition of the problem. In addition to providing the means of translating an English language description of a commercial process into an efficient program, it would be appropriate to provide a means of mechanically doing the detail work involved in machine syntax such as, memory layout, register logic, minimum access, optimum decision scheduling, and some of those other

jobs which, because of their very stochastic nature, will never be done efficiently by the human programmer in a large application.

It is necessary, therefore, to state in the problem definition critical information concerning the data. What are its major divisions of data for input and output? Card formats, fixed-character blocks, variable messages, printer line blocks? What are the subdivisions of data? One or more fixed-character words? One or more characters? In the RCA Bizmac approach,¹ the answers to these questions are available from standard data sheets. On these sheets (Fig. 1) each type of input data is indexed. The items are listed in the order of their serial appearance on tape or on a card, and the maximum number of characters which can appear in the item is shown. A place is left for listing the expected average item size since this gives a good measure of tape lengths in the Bizmac System, which in turn influence equipment to be used. Many of the computer instructions are also sensitive to actual (rather than maximum) item length.

(R) Stock Ledger

(name of data)

Description: A perpetual inventory and issue history record for each stock item

Order: Stock Number		Number of Messages (Peak):		Number of Messages (Normal):			
(1) Item Number	(2) Item Description	(3) Justify	(4) No. of Char. Max.	(5) Avg.	(6) % Use	(7) Weighted Avg.	
1	Stock Number	L	9	5	100	5	
2	Description	L	15	7	100	7	
3	Balance on Hand	L	8	3	100	3	
4	Total Issued to A	L	7	3	100	3	
5	Total Issued to B	L	7	3	100	3	
6	Total Issued to C	L	7	3	100	3	
7	Total Issued to D	L	7	3	100	3	
8	Reorder Level	L	8	3	100	3	
9	Unit Price (in dollars and cents)	L	7	3	100	3	
10	Procurement Lead Time	L	1	1	100	1	
Total Characters of Information			76	xxx	xxx	34	
Item and Message Codes (number of Items plus 2)			12	xxx	xxx	12	
Total BIZMAC Characters			88	xxx	xxx	46	

Fig. 1 - Bizmac reference data sheet

In addition to defining the formats of all the input and output data, it is necessary to order the inputs and outputs in relation to their dependence upon one another. For example, input 1 and input 2 might be required to produce output 1, and then input 1 and input 3 and output 1 may be required to produce outputs 2, 3, and 4. With each of these outputs should be associated a problem flow chart or its analogous narrative code. This dependence relationship of inputs and outputs might be worked out by the machine by deriving these associations from the data item indices; but it is felt that, for the present, it is wiser to keep the criteria for problem definition broad enough to permit the development of a highly flexible symbolic language. In addition, the presence of some duplication provides a basis for later accuracy control.

¹W. K. Halstead, J. W. Leas, J. N. Marshall, and E. E. Minett, "The Purpose and Application of the RCA Bizmac System"; A. D. Beard, W. K. Halstead, and J. F. Page, "Functional Organization of Data in the RCA Bizmac System"; A. D. Beard, L. S. Bensky, D. L. Nettleton, and G. E. Poorte, "Characteristics of the RCA Bizmac Computer"; L. S. Bensky, T. M. Hurewitz, A. S. Kranzley, and R. A. C. Lane, "Programming the Variable-Item-Length RCA Bizmac Computer": Papers presented at the Western Joint Computer Conference, San Francisco, California, February 9, 1956

The last part of the problem definition concerns the flow chart description of the processes being performed on input data (and/or output data) to obtain a given output. Flow chart symbols to represent all the operations being performed on the data are available with their corresponding symbolic equivalents. Figure 2 illustrates the simple decision pieces of the flow chart symbols with their symbolic equivalent, actual instruction equivalent, and verbal category.

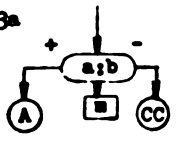
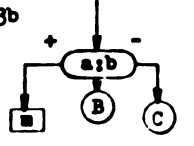
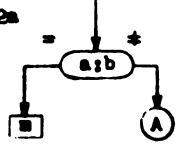
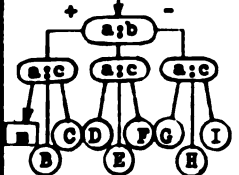
FLOW CHART DIAGRAM	SYMBOLIC EQUIVALENT	BIZMAC INSTRUCTION EQUIVALENT	VERBAL STATEMENT
1a  1b 	TC $\rightarrow$ A RTC $\rightarrow$ (A)	03 00 A 0000 0000 07 00 0000 0000 A	ONE-WAY CONTROL TRANSFER
3a  3b 	CTC (a.b) $\rightarrow$ A=C CTC (a.b) $\rightarrow$ mBC	15 02 b a 0000 02 00 A 0000 C 15 02 b a 0000 02 40 PC m00 C 03 00 B 0000 0000	THREE-WAY CONTROL TRANSFER
2a 	V(a.b) $\rightarrow$ mA	05 00 A a b	TWO-WAY CONTROL TRANSFER
	CTC (a.bc) $\rightarrow$ mB-I	15 02 b a 0000 02 44 m+ m+ m+ 04 00 MS 0000 0001 03 40 A 0000 0000 etc.	NINE-WAY CONTROL TRANSFER

Fig. 2 - The postulates of decision programming

An analysis of simple decision elements in a program indicated that they could be classified in much the same way as the elements of switching logic have been. The three-address instruction is ideally suited to a three-way decision, where two of the three branches are indicated by two of the addresses, and the remaining branch is the normal go-ahead or stopping function associated with all instructions. The actual criterion used for making the decision is the sign position of an arithmetic operation, where "0" can be sensed as the result for the third branch. If the three-address instruction is to be used for a two-way decision, one of the addresses must duplicate the control transfer. Symbolically, all of these conditions can be represented in neat formula form. For example

$$D(ab) \rightarrow mmC \quad (\text{two-way decision})'$$

means a decision is made depending upon the relation of $a + ob$ where, if $a < b$ or $a = b$, the processing continues with the next operation, and if $a > b$, the processing starts at a new point indicated by C.

In the same way, other decision types may be listed with their flow chart equivalents. Decisions depending upon deferred transfers of control and modified addresses are the most difficult to mechanize, and require a small subroutine.

An interesting effect on the overall running time is observed if the person defining the problem can include in his decision charting the expected high-volume flow path. On the basis of this additional problem information, the program layout can be changed to take the best advantage of the program access.

DEFINING THE MACHINE SYNTAX

Machine syntax is defined as the systematic arrangement of machine functions which determines instruction types and patterns irrespective of the problem.

Some of the machine characteristics which give rise to the necessity for special instructions are

- the difference in machine handling of alphabetic and numeric data,
- the machine method of addressing the data,
- the register requirements,
- the address structure,
- the method of termination of operations,
- the levels of storage for program and data,
- the manner of program and data input,
- the breakpoint control features,
- the required positions of the operands for an arithmetic operation.

As a specific example, Fig. 3 illustrates all those machine components involved in a program control transfer. This diagram was made as a result of studying all the machine components involved in making any decision in the program. The previous result indicator on the diagram contains the sign of the result of the last arithmetic operation, which is used in a conditional program control transfer. The program counter, subcounter, and surge-length register contain the proper exit points of program decision to the various levels of storage. The remaining blocks of the diagram are used in decisions involving referred program control transfers or address-modified control transfers.

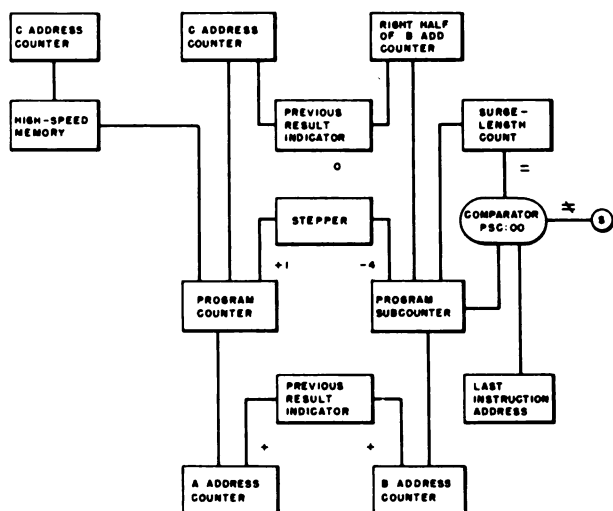


Fig. 3 - Block diagram for decision elements

In this important programming area, there is not a block on the diagram which is set directly from any information included in the problem definition. A case in point, and of particular interest in RCA Bizmac programming, is the description of the C counter function. One of the uses of the C addresses is to indicate the least significant character location of the result of an arithmetic operation. The operation may be chosen so as to terminate on a space symbol or on an item separator symbol. As the result gets transferred to the memory, character by character, the C counter which originally was set by the C address gets stepped each time a character is transferred. At the end of the operation, the C counter contains the position of the next consecutive memory address, which can be set up, that is, recorded in the high-speed memory. Essentially this provides the machine with the ability to keep track of where the data are in the high-speed memory. This is pertinent to machine considerations, but it is not desirable to require this in problem definition.

Other unique computer design features have important effects upon the type of problem to be processed and the relative extent of supporting machine syntax. Take, for example, the problem of programming a computer to scan plain language text for desired references, and assume that a minimum of input editing is desired. The machine syntax involved in handling this problem will differ unbelievably in type and amount with two different machines.

The RCA Bizmac computer has many unique machine characteristics, all of which affect directly the feasibility and ease of automatic programming. Some of the most important are:

1. The machine senses for control symbols in the data.
(No other machine has this feature.)
2. The high-speed memory (random-character addressable memory) affords a higher level of communication.
3. Certain operations may be terminated with space symbols. (In this case, the function of the space symbol in terminating arithmetic operations is to enable the computer to locate the positions of the words in a plain language message.)
4. Three-level storage, with two-channel parallel input, is available.
5. There is a two-bank, three-address logic.
6. Random-item composition is done on read-in and write-out.

The effects of such design upon programming might be appreciated by citing some of the programming areas that are eliminated.

1. There is no need for separate zero suppression since this is available as an operation option.
2. There are no extractor problems since all characters are addressable.
3. There is no requirement for double-precision operations since the adder output field is not limited.
4. There is a minimum of input editing.
5. Sorting, merging, and interrogation may be done on separate equipments; this is made possible by control symbols in the data.

In order to determine a basis for organizing the machine functions, a statistical study was made on programs of several major commercial applications. The frequency of occurrence of instruction sequences of two, three, and four instructions at a time was plotted. Figures 4, 5, and 6 illustrate the results of a relative distribution of occurrence of such instruction sequences. A study of these distributions indicated very good reasons for the prominent instruction patterns.

One significant case was the set 15 02 04 which was one of the high-frequency trigrams. This same set appeared in the analysis of the decision sets (see Fig. 2). By studying the high-frequency sequences of instructions, it was possible to develop a basis for defining these small "kernel" subroutines which were adaptable to generative techniques and by which one could program. Such programming might well be called metaprogramming.

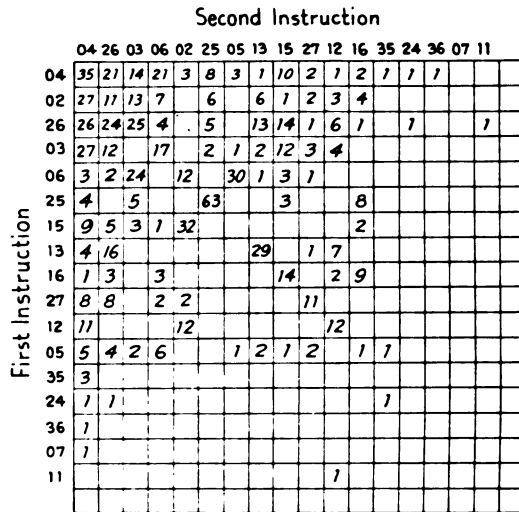


Fig. 4 - Scatter diagram showing instruction usage by pairs

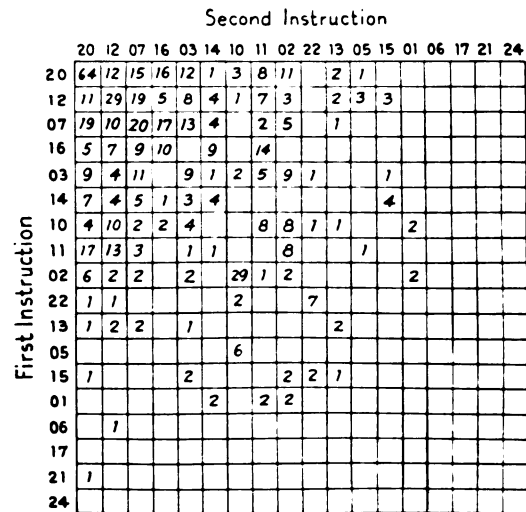


Fig. 5 - Cumulative pairs frequency

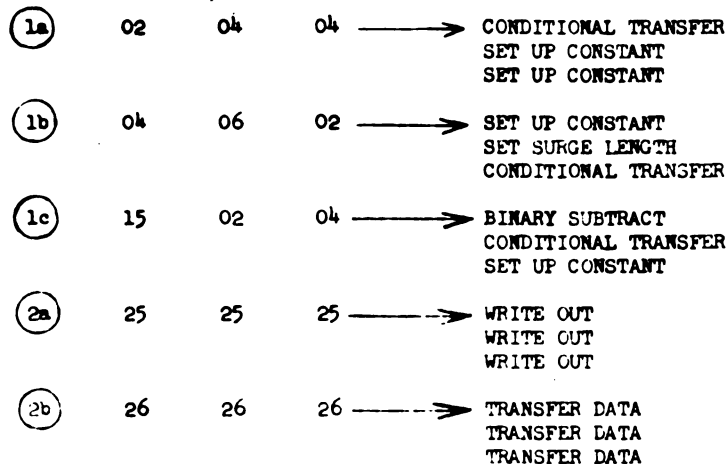


Fig. 6 - High-frequency instruction triplets

As would be expected, the organization of machine functions for automatic programming leads to the discovery of new design possibilities as a byproduct. It was observed, for example, that many of the metaprograms preserved a certain order of occurrence by function. First came one or more storing operations (in memory or registers), then a transfer of data, then an arithmetic operation, and finally, a decision operation. It was felt, therefore, that it would pay to assign the operation codes by these respective functions so that they might later be made

available by simple mathematical models (such as an arithmetic series) for generation. Designing for such operation code families provided an optimum basis for designing the gate drives, where the same function is to be executed under similar conditions in different instructions, since these instructions could be handled together on the same OR gate.

DEFINING METHODS FOR PROGRAMMING COMPOSITION

The integration of the problem functions and the machine functions to produce an efficient running program is done by passing the problem criteria through the computer as many times as is necessary to complete all the transformations on the problem criteria required to produce the final running program. Essentially, the method simulates the way the human programmer does the job, with the exception of the utilization of methods for making available known computer requirements and certain optimization operations. The human programmer uses a manual of computer instructions (or his memory) to fulfill the computer logic requirements. The machine uses "look-up tables" and "mathematical models" to provide the machine requirements. The former requires the correlation of machine function to machine language, hence the programming manual; the latter requires the correlation of machine function to problem language.

The first pass reads in the problem language or narrative code, which is the symbolic equivalent of the problem flow chart. This narrative code will be interpreted into computer problem operations with relative control transfer points and relative data references. Precedence relations will be used to order the narrative code pieces.

The second pass reads in information from the standard data sheets. The relative data references are entered into the memory allocation table together with the maximum item size and anticipated memory assignment for certain control purposes.

The third pass picks up the instructions derived from the problem and — by referring to the operation desired, the associativity to its neighbors and the machine functional dictionary, and the memory allocation table — successively extends the instructions to incorporate the necessary machine-derived instructions.

The fourth pass assigns the data addresses and adds standard program control features, such as accuracy control transfers, breakpoints.

The fifth pass assigns the program control addresses and edits the program for actual running conditions.

Problems of optimization involving data layout and program layout (minimum access) and those involving simultaneous operations and decision sequencing (scheduling) are in varied states of solution. The access problem is resolved. Some of the scheduling problems are resolved, others are being studied. One of RCA's engineers proposed a scheme for mapping all the program decisions in Boolean symbols on a two-dimensional matrix where the abscissa represented the order of the inputs, and the ordinate, the order of the outputs. The scheme appears to offer a method for program optimization with respect to the decision planning in the program, and is presently being studied as a programming tool.

Many of the programming methods used in this work involve the principle of generation. A recent paper in the I. R. E. Proceedings² states that: "A generator is a special type of compiler designed for data processing. If only a limited number of operations are to be performed upon each item of a large amount of data, the required program may be generated rather than pieced together from subroutines. The subroutines are reduced to 'coded coding.'"

It is seen that the saving concerns storage, and that, in place of storing all the possible little subroutines (metaprograms), the programmer is willing to pay in time, since he is not under any severe time restriction in machine program writing. How does he convert this space

²G. M. Hopper and J. W. Mauchly, "Influence of Programming Techniques on the Design of Computers," Proc. I.R.E. 41:1250 (1953)

to time? He does it by taking advantage of functional relationships between two of three sequences of numbers. The number sequences are, in machine language, the memory addresses, the memory contents, and the operation codes. This functional relationship may be a simple counting relationship, as in the case of a look-up table; it may be an arithmetic series relationship, or a geometric series, or a difference function. In any case, the programmer is utilizing one set of numbers to derive the other.

In one case, the memory addresses are related to memory contents.

In another case, the operations are related to memory addresses.

In a third case, the contents are related to operations.

In all cases, the programmer is utilizing some method of counting through one set of numbers to recover the other desired set of numbers.

For illustration, let us assume that the numbers stored in memory positions 1, 2, 3, 4, 5, 6, represent a meaningful sequence of instruction operations. Assume also that 2, 4, 6; 1, 3, 5; 3, 4, 5, 6; etc., represent meaningful sequences of instruction operations. In this case, instead of storing these instructions as metaprograms it would be possible to use the arithmetic series to generate them. Thus, 12 34 56 would be indicated as a two-digit number 11, where the most significant digit was the first term in the series, and the difference was 1. The criticism is obvious: What chance is there that these sequences of numbers will have meaningful computer significance? The question can be turned around, however, by asking: If not, what method of counting do you need to obtain meaningful sequences of numbers? In other words, if you can describe a counting method that produces for you your meaningful sequences of instructions, the technique could prove a valuable savings in storage requirements.

At RCA, we have discovered a new method of counting which promises some interesting developments in generator applications. This method of counting involves the definition of the binary-digit relationships with respect to a two-dimensional field of a sequence of numbers. This method will be the subject of a future paper.

These techniques depend upon the utilization of fundamental mathematical models. It is proper to mention here that these mathematical models generate many sequences of numbers not all of which have functional significance, but this is not of importance. The relevant aspect is that all of the sequences so produced are determined by the same counting mechanism or functions, and that only a key need be used to recover the desired sequence.

In conclusion, I would like to acknowledge the assistance of many of my colleagues in this field both at RCA and elsewhere; especially the discussions with Sidney Kaplan of RCA, and Professor Haskey B. Curry of Penn State University. One of Professor Curry's most significant comments³ on programming is: "You ask why I assumed an infinite memory in my analysis. It is the usual practice of the mathematician to proceed from the simple to the more complex. In this case, the infinite memory represented the simplest case. For finite memories, man-made constraints lead to more complex considerations. I have only attempted to show the way. What I see evolving is a new calculus of logic, that is programming logic. It is the responsibility of those in the field to develop this new calculus."

³H. B. Curry, "The Composition of Programs for Automatic Computing," Naval Ordnance Laboratory Memo 9005, January 26, 1949

THE PACT COMPILER FOR THE 701

R. G. Selfridge

*Naval Ordnance Test Station
China Lake, California*

Since this symposium is on advanced programming methods, I think I might well assume that the PACT system is at least a familiar name to many, if not most, of you. But in order to give a reasonably full description of the system it is worth giving some historical background.

IBM 701's started being delivered in 1953, and within a few months of delivery at each installation it became obvious that the expected easing of the time problem was never going to appear, and that actually it was only a matter of a few more months before these machines weren't going to be large enough. Hence, we were back to the old problem of shortening the time between appearance and solution of a problem. This led to a number of new assembly programs and other short-hand methods of putting a problem together.

By the summer of 1954 several of us were talking in terms of a compiler that would do far more of the work than available methods could do. At about this time the first information on FORTRAN became available, indicating that IBM had also been thinking on these terms, though for a nonexistent machine.

About November of 1954 Jack Strong and Frank Wagner of North American Aviation made the suggestion of a joint effort to produce a system. The original meeting was held to see if this made sense, and, if so, whether the different groups would be prepared to supply the necessary manpower. While there were some changes, the group that produced the first effort finally stabilized as Lockheed, Burbank; Douglas, Santa Monica, El Segundo, and Long Beach; North American Aviation; Rand Corporation; and The Naval Ordnance Test Station, China Lake. Since that time, IBM, General Electric, and Lockheed Missiles have come in, with the recent addition of Douglas Missiles.

At the start, our meetings ran several days a week and consisted of everyone shouting down everyone else's programming methods. Despite the names, we continued to speak to each other, and if one looks back at the notes from that period there was progress of a sort. In any case, by the spring of 1955 the outline of a system had been drawn up and coding was in progress. By June a deck of cards was put together, which naturally didn't work. But by August programs were being turned out which worked, and by last December one could say that the bugs had been worked out of the system.

What, perhaps, has been learned from all this? Several things of considerable value, I think. Foremost in many people's minds is the fact that several different organizations can get together and produce a joint offspring of considerable value. This has already appeared elsewhere in the form of SHARE.

Without much question such joint undertakings are of tremendous value, and are becoming even more valuable. But there are some aspects that may not be immediately apparent. In the first place, suppose one considers the economics of the problem. Nearly full time was spent by at least one person from each of seven organizations for about eight months. Then varying additional amounts were spent depending on the particular problems involved. Since more than one person contributed in each place, my own estimate of the time is eight to ten

man-years of work, though I should add that is higher than many other estimates. And on top of this is a considerable amount of machine time, keypunching, and the like, and the nasty little item that these ten man-years were taken from the best people each group had. There are very few companies that can afford that kind of an investment, especially in view of the scarcity of people.

One other advantage of a joint group is the widely divergent viewpoints one can get. Many of the best ideas that were included in PACT came as a result of A calling B's pet scheme worthless because This is far more likely if A and B do not work for the same group, because then their whole line of thinking differs. There are attendant disadvantages. If the group is too large, there is never enough overlap for any agreement, and the physical problem of space and keeping the meeting under control can become horrible. But by and large if such a group can come up with a system that is fairly satisfactory for all members, there is a far greater chance of its satisfying nonmembers, even though everyone has his own pet "improvements." One might add here that the group as a whole must learn to be ruthless to prevent such "improvements" or within weeks there will be n different systems.

All is not sweetness and bliss with such a group. The physical problem of getting together gets to be quite a nuisance, especially as the system begins to take shape, when only short joint periods are at all necessary. Another major difficulty is in the shape of the resulting system. But before I go into that, let me describe in more detail what resulted.

Briefly, what we had been given as a problem was to find some way of writing down equations that was quicker and more efficient than previous methods, with the machine handling the translation, assembly, and construction if needed. Machine language, whether symbolic, relative, or absolute, had many disadvantages. Could a more efficient language be produced?

What came out of this effort, and I might add this is only one of many possible ways we could have gone, was a system which allows coding that is far closer to mathematical expressions than any previous system (there are exceptions to this statement such as one system for Whirlwind). The problem of storage assignment is handled automatically, with two levels, permanent and temporary. Subscripts are allowed, with certain operations on the subscripts. These subscript operations, which I think is one of the real advances of our effort, include choosing a particular value for that subscript by equating to a variable, and automatic incrementing for generation of loops. Loops, in particular, become quite easy. A starting value is chosen, which may be a variable, a final value is chosen, which may also be a variable, and the compiler will set up the necessary setup, modification, and test instructions for the loop.

If I slide over the other features of the system there is still enough for my point. With the subscripts and storage assignment a large proportion of the bookkeeping is taken off the shoulders of the coder.

Having decided on this system, it still had to be made to work. Here the fact that we were a group from different locations made it far more desirable to break the coding into quite distinct pieces. This we found we could do, and the first stage of the coding was done by each group quite independently. Then, of course, the intercommunication had to be handled, and some of our advantage had to be paid back. Perhaps the best example of what can happen is to describe what arose not too long ago when working for the 704. Two consecutive sections, each fairly well checked, were put together and run. This stack of 50 to 80 binary cards succeeded in clearing the memory beautifully, and that was all.

But suppose the sections are finally put together. PACT for the 701 has, or had at the last estimate, a conservative 16,000 instructions, plus as long a library tape of subroutines as one might feel like inserting. If we had been from one place, all working together all the time, this might well have been cut by 20 to 30 percent. It is also possible that a few of the rules, which were constructed because of compiler coding problems, could have been loosened.

Perhaps enough has been said to show that there are distinct advantages to both methods of production. I am convinced, and I think the other members of PACT are also, that this group effort with different organizations has ample advantages to outweigh the disadvantages.

Other things have been learned from PACT besides this group problem. One that you might not suppose needs stating is that there must be some guarantee of permanence of the individuals making up the group. But the facts will show that PACT was delayed by turnover of people, and its 704 version has had even more difficulty. It is difficult to explain to people who have not tried, just how hard it is to take over someone's problem, even when a fairly complete description is available. It is far worse when it is a problem with a great many decisions to be made, and not just a solution of an equation. The result can be that it takes months to correct a fairly "simple" error, most of which time is spent discovering "why on earth does he do that." Life is not simplified by the fact that the high-caliber person who is needed by such a project is also needed very badly in myriad other ways.

Now perhaps one might draw some conclusions from the use of PACT. Its reception has ranged from "cool" to "it's a godsend." To some extent this is a function of the type of operation that each installation permits. There is a perpetual argument between open and closed shop, with all its different shades. The closed shop, where programming and coding a problem are handled by a specific group of people who rarely have any part in the generation of the problem, usually is not as satisfied with PACT as is the open shop, where the problem generator is often also the coder. This is understandable to the extent that the closed-shop coder is more experienced in coding and thus less in need of what PACT can offer. Another part of the problem is one of selling. It is always hard to get a new system into use, and in this case there were added obstacles. A new and untried system has to be introduced, and when it still has faults, as had PACT, it is extremely hard to persuade coders who have a working system with which they are completely at home that they should change. In fact, it can be stated with considerable emphasis that any new system of this type must not be introduced, except to a very select group, until it has been very thoroughly checked out.

An open shop, in contrast to a closed shop, usually has a continuing flow of people learning to run their problem through the computer. Here the introduction of a new system is far simpler, and the presence of faults in the new system is no more troublesome than the faults the new coder is having anyhow. Also, if the system does not have a fault for that particular problem, the coder is far better off, for he can get his problem running far more rapidly. The statistics for such arguments have never been put together, but there are many cases where unexperienced people have had their solutions in a matter of days, and not on trivial problems either.

Before one decides that this is for the novice, let's hold on a moment. There are plenty of very experienced coders who much prefer PACT if it can be used (and conversely, I regret to say). Here I think of it as being a question of aim. The coder whose main aim is to solve a problem, rather than to solve a coding problem, is more likely to buy such a compiler. The coder who spends all his time coding will be efficient enough that PACT may not gain him enough to be worth its cost. But there was one memorable case where a very competent coder had not been able to make a problem run, which PACT handled from start to finish in a couple of hours. This difficulty arose from the extreme interturning of loops which caused all sorts of address modification troubles.

This range of opinions can be found right in the group that generated PACT, without considering any outsiders. Include outsiders and the opinions may vary even more. However, I think that most of the people who have used PACT enough to decide whether or not they like it will agree that it does perform certain jobs quite well. The construction of loops is quite effective, and the allocation of storage, permanent and temporary, is a considerable convenience. A problem written in PACT involves far less writing than does one written in machine language. The ratio of PACT instructions to final machine instructions has run from one to three, to one to ten, which is considerable. It is quite true that an experienced coder can beat PACT, but only if he is willing to take the time, or to start from the PACT production.

I think that the best evidence of the desirability of this system is that there was plenty of support for the changeover to a 704 system, and that the PACT group is still in existence, working on the next stage of the problem. To dive into the future a bit, it is the consensus of our group that the next problem to be solved is not directly that of compilers, but one that an efficient compiler makes far more important: the twin problem of secondary storage and debugging one's code. The introduction of a compiler between the coder and the machine has

forced some form of retranslator so that the coder stands a chance of finding out why his problem won't run. Our present hopes are to solve this in part before worrying about another compiler.

I think I can best end by pointing out that nearly all people in the computer field are convinced of the need for automatic coding, and that, with the introduction of multiple computing circuits, such automatic codes are not going to be simple, will have to do quite a lot of program searching before compilation, and are going to be even more vital than ever.

AUTOMATIC DIGITAL ENCODING SYSTEM II

E. K. Blum

*Naval Ordnance Laboratory
White Oak, Maryland*

In order to explain the ADES approach to automatic programming, it is useful to begin with some heuristic considerations. Let us ask, "What is the ultimate objective of automatic programming?" One answer might be that the ideal automatic programming system should be able to accept any problem formulated as a set of mathematical equations in some more-or-less conventional notation, and produce therefrom a complete program of instructions for any digital computer.

Pursuing this further, let us suppose that a problem, as presented to a digital computer group, consists of three parts:

1. the mathematical equations,
2. statements describing the input data, and
3. statements describing the desired results.

The mathematical equations can be of several kinds; they may include algebraic equations, differential equations, integral equations, etc. The input statements specify numerical values to be assigned to parameters, thereby determining the number of different cases to be computed, and they specify also which variables are to be regarded as independent variables for which the range of values is known. The remaining variables are treated as dependent variables, that is, as quantities to be determined by means of the equations. The output statements then specify which of these computed quantities are to be printed as final results.

In this form, the problem is not susceptible of digital computation. First, the variables are "continuous"; that is, they are assumed to range continuously over some interval. Second, operations like differentiation and integration are nonarithmetical and can only be approximated by a digital computer. At this juncture, the problem is usually subjected to numerical analysis, the continuous variables being replaced by discrete variables. Thus, an independent variable, x , which was supposed to range continuously over the interval $[a,b]$ might be assigned a discrete set of values, x_i , where i is to take on integral values from 0 to n , say. The specification of the range of the index, i , will be called quantification.

Next, implicit equations, if any, are replaced by explicit equations by applying some numerical method such as Newton's method. This usually involves an iterative procedure and again a quantification is called for. Finally, derivatives are replaced by differences, integrals by sums, and the numerical formulation is passed on to the programmer.

This is considered to be the starting point for ADES. It will accept the numerical equations, together with the input-output specifications and the quantifications, and translate them into a computer program, compiling subroutines and automatically assigning addresses and operation codes in some language comprehensible to the particular computer.

Of course, this oversimplifies the situation considerably. It should be obvious that the numerical formulation must be written in some standardized notation. Hence, the first and most important component of ADES is a formulation language, and most of what follows will be devoted to a description of this language. This description will be semiformal, consisting of definitions and examples.

The ADES language is essentially mathematical in structure. It is based on the theory of recursive functions and the schemata for such functions, as given by Kleene.¹ The alphabet of the language is constructed from the nonnegative integers and the following nine generic symbols:

a		independent variable symbol
b		dependent variable symbol
c		free variable symbol
q		independent index symbol
r		dependent index symbol
f		function symbol
e		equal sign symbol
p		punctuation symbol
d		output symbol

The usage of these generic symbols is indicated by their names. A letter of the alphabet consists of a generic symbol with a two-digit subscript attached. Thus, a_{00} , b_{01} , a_{23} , f_{23} are letters.

The letters a_{00} , a_{01} , a_{02} , ..., a_{99} are called independent variables. The letters q_{00} , q_{01} , q_{02} , ..., q_{99} are independent indices, and so on.

The simplest kinds of expressions which can be formed with the letters of the alphabet are the indexed variables defined as follows:

Definition 1 - Let x denote one of the independent or dependent variables, a_{00} , a_{01} , ..., a_{99} , b_{00} , b_{01} , ..., b_{99} . Let i and j each denote one of the indices q_{00} , q_{01} , ..., q_{99} , r_{00} , r_{01} , ..., r_{99} . An indexed variable of degree one is an expression of the form xi . An indexed variable of degree two is an expression of the form xij . For convenience, a variable, x , will be called an indexed variable of degree zero.

What is intended here is that a one-dimensional array of data should be denoted by an indexed, independent variable of degree one; e.g., $a_{03} q_{12}$. Likewise, a two-dimensional array of data (i.e., a matrix) would be denoted by an indexed, independent variable of degree two, say, $a_{23} q_{11} q_{12}$; and so on.

Definition 1.1 - A numerical constant is a floating-point number written in some standard form consisting of an exponent, modulus, and algebraic signs. (We are assuming that the computer in ADES can perform floating-point arithmetic. In fact, all operations will be carried out with floating-point arithmetic. Since the precise structure of a floating-point number depends on the particular computer, which we do not specify, we shall write numerical constants in the form that is most convenient.)

Next in the hierarchy of expressions are the terms of the language. To define "term" we must first recall the definition of a well-formed formula as used in mathematical logic. This may be stated recursively as follows:

If x is an operand, then x is a well-formed formula. If $y_1, y_2, \dots, y_n$ are well-formed formulas and ϕ is a function of n arguments, then $\phi y_1 y_2 \dots y_n$ is a well-formed formula.

¹S. C. Kleene, "Introduction to Metamathematics," New York:Van Nostrand, 1952

Definition 2 - An index term is a well-formed formula in which the operands are indices, indexed independent variables, or numerical constants.

Definition 3 - A b-term is a well-formed formula in which the operands are indexed variables or numerical constants.

Definition 4 - An r-equation is an expression of the form

$$j = \theta,$$

where j denotes one of the dependent indices, $r_{00}, \dots, r_{99}$, and θ denotes an index term.

A b-equation is an expression of the form

$$y = \psi,$$

where y denotes one of the dependent variables, $b_{00}, \dots, b_{99}$, and ψ denotes a b-term.

The b-equations and the r-equations constitute the major part of the formulation of a problem, since they define what is to be computed. In the ADES language, the terms in the equations must be written in the parenthesis-free notation. In this concise notation, each function symbol is written to the left of its operands and all parentheses are eliminated. This will be illustrated by several examples. In writing these examples, the conventional mathematical symbols $+$, $-$, $/$, and $\cdot$ will be used for the arithmetic operations. This is done for expository reasons only, it being understood that when actually writing a term in ADES language, the formulator must write f_{02} for $+$, f_{03} for $-$, etc. Similarly, the conventional equal sign is used in the examples.

Now, to illustrate, we write a list of "conventional" mathematical equations on the left and the corresponding ADES equations on the right.

$r_{01} = q_{01} - a_{01},$	$r_1 = - q_{01} a_{01},$
$r_{12} = [a_{01}(q_{02}) + r_{02}] / q_{07},$	$r_{12} = / + a_{01} q_{02} r_{02} q_{07},$
$b_{04} = [a_{01}(q_{02}) + a_{03}] b_{03},$	$b_{04} = \cdot + a_{01} q_{02} a_{03} b_{03},$
$b_{02} = b_{05}(r_{01}, r_{02}) - a_{01}(q_{01}, q_{02}),$	$b_{02} = -b_{05} r_{01} r_{02} a_{01} q_{01} q_{02} \cdot$

The occurrence of an independent index in an equation requires that a quantification be written to specify the lower and upper bounds for this index. The correct placement of this quantification will help to determine the flow chart of the problem. In ADES, a large part of the flow chart is determined implicitly and in a natural way by the very structure of the b-equations. However, certain explicit directions must be written to quantify indices, to indicate branch formulas, and for recursions. These will now be explained briefly by definition and example.

Definition 5 - A quantification is an expression of the form

$$P_{03} L i u$$

where P_{03} is a punctuation symbol which denotes the universal quantifier "for all," i denotes an independent index, L denotes the lower bound of i and may be an index or an indexed independent variable, and u denotes the upper bound of i and may be an index or indexed independent variable.

The quantification, $P_{03} L i u$, is to be read as "for all integral values of i such that $L \leq i \leq u$."

The writing of quantifications in an ADES formulation is governed by several syntactical rules. The first rule is as follows:

Rule 1 - Precisely one quantification must be written for each independent index which appears in a formulation. The quantification is written to the left of one of the b-equations. Unless explicitly forbidden by other rules, any b-equation can be preceded by one or more quantifiers. The order of quantification is from left to right.

As an example, suppose we wish to compute $a_{01}(q_{01}) + a_{02}(q_{01})$ for all integral values of q_{01} from 0 to 5. Designating the sum by b_{01} , we would write

$$P_{03} \ 0 \ q_{01} \ 5 \ b_{01} = + \ a_{01} \ q_{01} \ a_{02} \ q_{01}, \ .$$

Again, if we wish to compute $a_{01}(q_{01}, q_{02}) - a_{02}(q_{01}, q_{02})$ for all $0 \leq q_{01} \leq 5$ and $0 \leq q_{02} \leq q_{01}$, we write

$$P_{03} \ 0 \ q_{01} \ 5 \ P_{03} \ 0 \ q_{02} \ q_{01} \ b_{01} = -a_{01} \ q_{01} \ q_{02} \ a_{02} \ q_{01} \ q_{02}, \ .$$

If it is necessary to quantify several b-equations by the same quantification, or if it is necessary to specify the order in which several independent quantifications are to take place, a special kind of b-equation, called a phase equation must be written.

Definition 6 - A phase equation is a b-equation of the form,

$$y = f_{00} y_1 y_2 \dots y_n,$$

where the y 's denote dependent variables and f_{00} is a special function. The function, f_{00} , is to be read as "program the equations which define $y_1, \dots, y_n$." Thus, a phase equation is a metamathematical statement rather than a mathematical one.

A phase equation in which the left member, y , is the special dependent variable b_{00} , is called a master phase equation;

$$b_{00} = f_{00} \ y_1 \dots y_n, \ .$$

Rule 2 - There is precisely one master phase equation in each problem formulation. The dependent variable, b_{00} , is used only once in the formulation. The master phase equation is the first equation to be programmed.

Rule 3 - To quantify several b-equations by the same quantification, a phase equation is written as

$$P_{03} \ L \ i \ u \ y = f_{00} \ y_1 \dots y_n, \ .$$

This means that the formulas which define $y_1, \dots, y_n$ are to be computed for all values of i from L to u .

For example, if it is required to compute both $a_1(q_1) - a_2(q_1)$ and $a_3(q_1) + a_5(q_1)$ for all values of q_1 from 0 to 9 inclusive, the formulator could write the equations:

$$P_{03} \ 0 \ q_{01} \ 9 \ b_{00} = f_{00} b_{01} b_{02},$$

$$b_{01} = -a_{01} q_{01} a_{02} q_{01},$$

$$b_{02} = +a_{03} q_{01} a_{05} q_{01}, \ .$$

Each quantification specifies a loop in the flow chart of the computation.

Now, to provide for dependent variables which are to be computed by one of two or more alternative formulas depending on specified conditions, we introduce the following definition:

Definition 7 - A branch equation is an expression of the form,

$$y = \phi \ x_1 \ x_2, \ \psi, \ \theta,$$

where y denotes the dependent variable being defined; ϕ is a function denoting one of the three conditions $<$, $\leq$, and $=$, respectively; x_1 and x_2 are variables; ψ denotes the b-term which defines y if the condition $\phi \ x_1 \ x_2$ holds; and θ denotes the b-term which defines y if the condition does not hold.

As an example, suppose we wish to define a quantity which is equal to $a_{01}(q_{01}) + a_{02}(q_{01})$ if $a_{02}(q_{01})$ is less than 0 and equal to 1 otherwise. We would write the branch equation

$$b_{01} = < a_{02} q_{01} 0, + a_{01} q_{01} a_{02} q_{01}, 1, .$$

An analogous definition is used to define a branch r -equation for dependent indices. Multiple branch definitions are obtained by repeated use of the branch equation.

Perhaps the most complicated part of the syntax of the ADES language is that which deals with recursion. Many different types of recursions are provided for, and it is not possible to describe them all here. I shall mention two types, illustrating them by elementary examples. For a complete discussion, NavOrd Report 4209² should be consulted.

The Newton algorithm for the square root of a_{01} can be formulated as follows:

$$\begin{aligned} 0 \leq i \leq 5 \quad x(i+1) &= [x(i) + a_1/x(i)]/2, \\ x(0) &= 3. \end{aligned}$$

This is a simple recursion, since only one index, i , is involved. This formulation calls for six iterations. Rewriting the formulation in ADES language, we obtain,

$$\begin{aligned} b_{00} &= f_{00} b_{01}, \\ P_{03} 0 q_{01} 5 b_{01} &= 11 / + b_{01} q_{01} / a_{01} b_{01} q_{01} 2, 3, \end{aligned}$$

where the initial value, 3, is set off by commas; q_{01} is used as the recursion index; and $b_{01} q_{01}$ denotes the iterant at the q_{01} stage.

A double recursion involves two indices. As an example, we take the formula for the binomial coefficients. Denoting the coefficients by $b_{01}(i, j)$, we have

$$\begin{aligned} b_{01}(i+1, j+1) &= b_{01}(i, j+1) + b_{01}(i, j), \\ b_{01}(i+1, 0) &= 1 \quad \text{for all } i \geq -1, \\ b_{01}(0, j+1) &= 0 \quad \text{for all } j \geq 0, . \end{aligned}$$

To rewrite this as an ADES formulation, we replace the row index, i , by q_{01} and the column index, j , by q_{02} . Thus, we have

$$\begin{aligned} r_{01} &= + q_{02} 1, \\ P_{03} - 1 q_{01} a_3 (P_{03} - 1 q_{02} a_3 [b_{01} = 12 < q_2 0, 1, b_{02},]) \\ b_{02} &= < q_1 0, 0, + b_{01} q_{01} r_{01} b_{01} q_{01} q_{02}, \\ b_{00} &= f_{00} b_{01}, . \end{aligned}$$

In the theory of recursive functions, an operator known as the minimization operator is introduced. This is incorporated into the ADES language in the following definition:

Definition 8 - A minimization equation is an equation of the form

$$j = f_{\mu} i x y,$$

²E. K. Blum, "Automatic Digital Encoding System II (ADES)," Naval Ordnance Laboratory Report 4209, February 8, 1956

where j denotes a dependent index, i denotes an independent index, x denotes an indexed independent variable or constant, y denotes a dependent variable, and f_{μ} denotes the minimization operator. The equation can be read as " j is the minimum value of i , $i \leq x$, such that $y \leq 0$." A more complete treatment of minimization is given in NavOrd Report 4209.³

Having written the equations and quantifications, the formulator completes the formulation by writing input-output specifications. Again, we refer to Report 4209 for details. In brief, the input specification consists of a table of the independent variables used in the formulation versus the maximum number of data to be supplied to the computer for each such variable. The final output specifications consist of appropriately subscripted d -symbols written to the left of the pertinent b -equations.

I shall conclude with a few remarks about the second component of ADES, the translating device, called the Encoder. The logical design of the Encoder, completed some time ago, is independent of the computer. It is only at the final translation step that the Encoder must take cognizance of the particular computer being used. A simplified version of an Encoder for use with the IBM 650 calculator is now nearing completion. Results of our experience with this model should be available by the end of the summer.

³Ibid.

ON A PROPERTY OF NATURAL LANGUAGE
AND ITS USE FOR THE DESIGN
OF IMPROVED MACHINE LANGUAGES
(ASSOCIATIVE MACHINE LANGUAGES)

Robert Serrell

*Radio Corporation of America
David Sarnoff Research Center
Princeton, New Jersey*

It was remarked at the Eastern Joint Computer Conference last November that the development of modern programming methods could be much advanced by the improvement of machine languages. There is probably no single best machine language for all purposes; but, for the best development of advanced programming techniques, the machine languages of the future should possess many useful properties that they do not now possess. Two of these properties, for example, readily come to mind:

First, machine languages should be designed to permit optimum utilization of internal storage capacity, and this with a constantly more complex definition of the word "optimum." In spite of important improvements in electronic storage equipment, the effectiveness of modern computers is often limited by internal memory capacity. Mere memory enlargement does not suffice; problems grow faster than computers. Thus it is necessary to better utilize such memory capacity as can economically be secured.

Second, machine languages should be so designed that programming systems (of the interpretive, compiling, or other kinds) developed for one type of machine are readily adaptable for use with other comparable types. The adaptation should require nothing more than a simple machine process, like that of converting data from one number system to another. New programming methods developed for a particular machine type cannot at present be used with other machine types as readily as they should and deserve to be used.

There exist in our natural everyday language properties that are the result of long evolution and that, because of their origin, may be considered basic. An interesting one of these properties is a relationship between the relative frequencies of words and the numbers of distinct meanings of the words.

It was noticed at the RCA Laboratories a few years ago, while compiling data for the design of coding systems, that the more frequent words have a larger number of distinct meanings per word. Excepting two dozen or so most frequently used words that are all connectives (such as the, that, and, it) the relationship appears to exist over the entire range, from the most frequent to the least frequent words.

The words occurring in telephone conversations, concerning which sizable statistics are available,¹ provide a convenient illustration of this relationship. Figure 1, which treats words in sets because single words are not statistically significant, shows the average number of meanings per word plotted against the relative frequencies of occurrence of these words as

¹See for example: N. R. French, C. W. Carter, Jr., and W. Koenig, Jr., "The Words and Sounds of Telephone Conversations," Bell System Tech. J. 9:290 (1930)

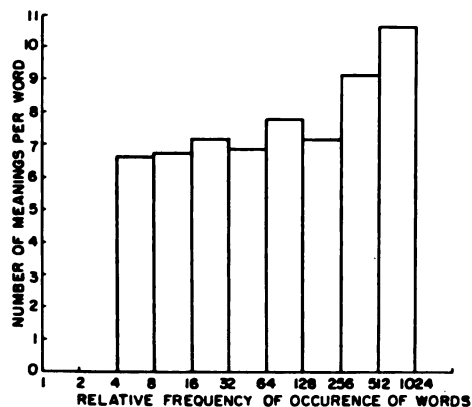


Figure 1 - Word statistics in telephone conversations

they are used in telephone conversations. The number of meanings of each word was found in a standard dictionary. Because the data were used in an application of information theory, the frequency scale is logarithmic to the base 2.

Note that the relatively most frequent words (aside from connectives which were left out), those occurring 512 to 1024 times in the normalized sample, have, on the average, more than ten meanings per word; while the words occurring 4 to 8 times in the normalized sample have only six to seven meanings per word.²

Much the same results were obtained from the "Semantic Count of the 570 Commonest English Words" published by Professor Irving Lorge (of Teachers College, Columbia University).

As pointed out by Professor Lorge in a private communication, the relationship exists because the meaning of a word is, in general, largely determined by the context in which it appears, that is, by its association with other words; and the more frequent words, because their appearances are more numerous, enter into a greater number of distinct associations with other words and, hence, have a greater number of distinct meanings.

Thus, the relationship between frequency of occurrence and number of meanings arises essentially from associativity among the words of a language. It cannot exist in a language the words of which have no associative meanings. In a language in which the relationship does exist, its extent and character provide useful measures of the extent and character of associativity within the language.

Now it is evident that most, if not all, machine languages in use today exhibit no associative meanings. In today's typical machine language, the meaning of every instruction word is unique, fixed, and quite independent of any association with other words in the language.

But it can be shown, with the help of the property discussed above, that the common words we use to describe the sequences of operations required to solve a given problem possess many associative meanings.

Consider, for example, the word "add." It may denote the addition of two numbers or of many numbers; it may denote algebraic addition or the addition of absolute values; it may denote the addition of vectors, or of matrices, etc. The word "multiply" evidently possesses a similar multiplicity of meanings. Notice that the words "add" and "multiply" both occur very frequently in computation work.

Consider now an expression such as "take the absolute value." In general, it occurs very much less frequently than "add" or "multiply." And it has just one mathematical meaning.

Thus it appears that, in the common language of numerical work, frequency of occurrence and number of distinct meanings are commensurate. It appears, therefore, that this language possesses associative characteristics. The language of applied mathematics may well be, in fact, one of the most associative of all languages.

²The calculations of the data from which these figures were taken were performed by Mrs. K. J. Rupprecht.

Insofar as the design of an improved machine language is concerned, the implication of all this is quite clear. The structure of the machine language should correspond to that of the language of computation work. That is to say, the more frequently used words should possess a greater number of distinct meanings, these meanings being obtained by explicit association of words within sentences in the machine language.

The incorporation of associative properties into a machine language should appreciably shorten the programs prepared in the language. Many meanings that are now supplied by words would be supplied by associations of words, so that fewer words would be required. There is little doubt that machine language programs would also be easier to prepare in that manner. An associative language is more natural and far more flexible. Further, since the computing machine should no longer be required, by the rigid structure of its own language, to perform purely paraphrasing operations such as the transfers imposed by address limitations, computation time would also be shortened.

An associative machine language is principally characterized by

Extensions of the meaning of the operation part of a machine instruction³ to correspond to mathematical meanings.

Unlimited address structure.

Use of connectives of various kinds to permit the formation of sentences of instruction words.

Let us briefly consider each of these characteristics in turn

1. It is obvious that it should not be necessary to repeat an operation word within any statement of its operands. It is sufficient, of course, that every operation to be performed be specified once, no matter how many operands are involved in it. The meaning conveyed to the computer is that it should perform whatever operation has been specified until a new operation is required.

Further, there exist in computation work regular sequences of arithmetic operations (such as sums of products, sums of quotients) that are very frequently used. Most of the common functions are evaluated by means of sequences of this sort. It should not be necessary to repeat the words "multiply," "divide," "add," etc., every time the corresponding operation occurs in a regular sequence. It should be sufficient to define each distinct operation required once, and then to define the sequence.

2. There is really no logical reason why the address structure of a machine language should be fixed. A smaller total number of addresses are required if the addresses in each instruction are not arbitrarily limited.

3. In an associative machine language, connectives are needed to indicate the nature and extent of the associations of words. Connectives (with the help of punctuation) permit the formation of associatively meaningful sentences.

To illustrate these ideas, I shall now describe a very simple associative machine language. The description⁴ consists of

a list of twenty-two words with their individual meanings (dictionary)

a punctuation rule

five syntactical rules

eight definitions of associative meanings.

It is assumed that the characters used to form words are each composed of five binary digits, so that an alphabet of thirty-two distinct characters is available. The number representation is assumed to be binary-coded decimal, with the ten decimal digits corresponding, for simplicity, to their pure binary (five-digit) equivalents.

³For the purposes of this paper, I shall define a machine instruction as a complete sentence composed of distinct instruction words.

⁴The Experimental Associative Machine Language is given at the end of the paper.

The numbers in parentheses represent single (five-binary-digit) characters. Numbers not in parentheses are composed of binary-coded decimal digits.

The advantages of using this associative machine language — as well as the manner of using it — can be demonstrated by means of a few simple problems. (A three-decimal-digit fixed address length has been assumed.)

1. Compute the second-order determinant whose first-column elements are in storage locations 341, 351 and second-column elements in locations 361, 371. One instruction:

(16) 341 371 (27) 351 (25) 361 (31)

2. Compute $a/b + c/d - e/f$, with the operands a, b, c, d, e, f in locations 400 to 405 respectively. Put the result at location 82 and clear register. Get the next instruction from location 641. One instruction:

(18) 400 401 (27) 402 403 (27) 404 (25) 405 (29) 082 (30) 641 (31)

3. Compute the algebraic sum of the numbers stored at locations 171 to 174. If this sum is larger than the number in location 942, stop. Otherwise, take the next instruction from location 943. Three instructions:

Computing the sum and
subtracting criterion: (11) 171 (28) 174 (25) 942 (31)

Conditional transfer: (20) (10) 943 (31)

Stop: (24)

4. Compute $ABCDE + FGHI - JKL$ with $A...E, F...I, J...L$ in storage locations 600...604, 605...608, 609...611, respectively.

One instruction:

(16) 600 (28) 604 (27) 605 (28) 608 (27) (25) 609 (28) 611 (31)

Note that since, in this example of an associative machine language, addresses are of constant length, they need not be separated. It is only for clarity that they have been separated here. Of course, the words of the language are separable from addresses through the fact that they are not composed of any of the binary-coded decimal digits 0...9, while all addresses other than that of the immediately preceding result are entirely composed of these digits.

Instead of addresses containing a fixed number of digits, it has been suggested that addresses of variable length be used, together with a suitable "space" symbol to separate them. But it is easy to show that, unless the short addresses appear very much more frequently than the long ones, no saving in the average length of instruction sentences can be obtained in this manner. If all decimal addresses appear with the same frequency, the average number of decimal digits per address is smaller than the maximum number by only

.1	digit for 10^2 storage locations
.11	digit for 10^3 storage locations
.111	digit for 10^4 storage locations
.....	
.11111	digit for 10^6 storage locations

etc.

With addresses of variable length at least one space symbol per address is necessary. So that, if these variable-length addresses are randomly distributed, the average length of the instruction sentences is increased rather than decreased by their use.

It is clear, of course, that insofar as complete instruction sentences are concerned, rather than only addresses, variable length does provide considerable savings of storage space.

Much of the power of modern computing methods is derived from the automatic modification of stored instructions by the computer itself as the computation proceeds. Most commonly, addresses only are modified, by arithmetic operations performed upon them. The use of an associative machine language involving connectives permits a useful extension of the automatic modification of instructions by the computer. Arithmetic operations can automatically be performed on connectives, as well as on addresses. As an example of the interesting possibilities this brings up, note for instance that, in the symbolism used a moment ago,

"Negative" + 00001 = "Absolute"
"At" + 00001 = "Next"
etc.

To function in an associative language of the kind I have described, a computer would require special logical circuitry. This circuitry would consist mainly of a separate, additional accumulator, means to store variable-length instructions, and special program registers for the formation of associative meanings.

The additional accumulator surely presents no serious difficulties. Concerning the second item, it is obvious that the circuitry required for variable-length storage is more complex than for fixed-length storage. But this additional complexity appears to be an economic necessity. It should be noted that a considerable amount of machinery using variable-length storage is in operation now.

Insofar as the special program registers are concerned, much depends upon the syntax of the associative language used. Note that, in the example I have given, the operation word (or verb) is the first word in every instruction sentence. This arrangement permits the machine to execute each instruction word by word, as the operands are transferred. The whole sentence need not be stored all at once in a program register. Of course, the associative language that I have described is a very rudimentary one. With more general syntactical rules, some special program registers will be necessary for the formation of relevant contexts.

To conclude, let me very briefly point out how the use of an associative machine language can provide the improvements mentioned at the beginning of the paper.

The sample instruction sentences demonstrate the extreme conciseness of associative machine languages. This conciseness is the result of the structural organization of the language: the more extensive the organization, the more concise the language. It follows that the effectiveness of memory utilization which results is an organic one. It grows just as problem complexity grows.

With an associative language, a computing machine can function in nonassociative modes. In particular, it can function in any fixed multiple-address code that happens to be convenient for a given purpose. Thus, it appears that such a computer can utilize programming systems designed for another machine of comparable (but nonassociative) type, with little more than a simple transliteration of the latter's basic language.

I would like to say that, as in previous instances, I owe much to my coworkers in the Radio Corporation for their part in the development of the ideas I have mentioned.

EXPERIMENTAL ASSOCIATIVE MACHINE LANGUAGE

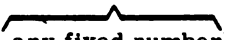
Word List

Operation Words

- (11) Add algebraically all the operands whose addresses follow
- (12) Add the absolute values of all the operands whose addresses follow
- (13) Subtract from the operand whose address follows (minuend) the operand at the second address (subtrahend)
- (14) Same as (13) but with absolute values of the operands
- (15) Multiply together all the operands whose addresses follow
- (16) (Multiply accumulate) Form the product of the operands whose addresses follow, transfer it into accumulator and clear arithmetic unit, repeat with operands following "also"
- (17) Divide the operand whose address follows (dividend) by the operand at the second address (divisor)
- (18) (Divide accumulate) Form the quotient of the operands whose addresses follow, transfer it into accumulator and clear arithmetic unit, repeat with operands following "also"
- (19) Transfer the result of the immediately preceding (arithmetic) operation to the address which follows, leaving the preceding result in its register
- (20) (Conditional transfer) If the number stored at the first address is negative or zero, take the next instruction at the second address; otherwise, continue in the specified sequence
- (21) Same as (20) but "... positive or zero..."
- (22) Store input information at the address which follows
- (23) Supply output information from the address which follows
- (24) Stop

Addresses

- (10) Address of the immediately preceding result
- (xx...x) Operand address, or address to which the preceding result is to be transferred, or instruction address

any fixed number
of binary-coded
decimal digits

Connectives

- | | |
|------|----------|
| (25) | Negative |
| (26) | Absolute |
| (27) | Also |
| (28) | To |
| (29) | At |
| (30) | Next |

Punctuation

- | | |
|------|-----------------------------|
| (31) | End of instruction (period) |
|------|-----------------------------|

Punctuation Rule

Every instruction other than "Stop," regardless of its length, ends with a period.

Syntactical Rules

Every instruction begins with an operation word.

There is only one operation word per instruction.

The connective "To" should be used only between addresses.

When either of the connectives "At," "Next" is used, it should be placed after all operand addresses.

When both connectives "At," "Next" are used in an instruction, "At" should appear before "Next."

Definitions of Associative Meanings

The meaning of every arithmetic operation word other than "Subtract" or "Divide" extends over all operands whose addresses are either unseparated from the operation word or separated by "Negative" or "Absolute."

When the operation is a subtraction or a division, the second address is that of the subtrahend or of the divisor.

The connective "Negative" causes the operand whose address immediately follows to be multiplied by -1 before it is used.

The connective "Absolute" causes the operand whose address immediately follows to be taken with its absolute value.

The connective "Also" causes an "Accumulate" operation to be performed again on all the operands whose addresses follow the word "Also." (Whenever an "Accumulate" operation is performed, the final result is taken from the accumulator.)

The connective "To" designates any sequence of three or more consecutive addresses, the first and the last of which are given and which are all to be used in the same manner.

The connective "At" designates the address at which the result of the operation is to be placed. Using the connective clears the register of this result.

The connective "Next" designates the address of the next instruction whenever this instruction is not in an implied sequence.

To avoid fine, this book should be returned on
or before the date last stamped below

MAR 5 1959

~~SECRET~~ 15-278
Send to dept
COMPUTER SCI
LIBRARY 1162

~~SECRET~~

COMPUTER
SCIENCE

510.78
462
1921

~~MA~~
~~STANFORD~~
LIBRARY

Stanford University Libraries
Stanford, California

Return this book on or before date due.

JUN 3 1978

MAR 17 1977

